Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Diploma Thesis

Energy-Efficient Scheduling for Multi-Core Processors

The System Architecture Group
Prof. Dr. Frank Bellosa
University of Karlsruhe

Johannes Lieder

Advisors:
Prof. Dr. Frank Bellosa
Dipl.-Inform. Andreas Merkel

November 18th, 2008

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 18. November 2008 Johannes Lieder

Abstract

Constant advances in processor microarchitecture promise new levels of performance and efficiency with each new hardware generation. However, depending on the type of software a computer is processing, running the CPU at the highest frequency setting may not be the most efficient mode of operation. Previous work in the area of power management demonstrated that it is possible to make an operating system energy-aware in order to run hardware at its most efficient performance state. With the emergence of multi-core processor architectures, new challenges arise for operating system software and CPU power management in particular. The main obstacles are caused by hardware dependencies between cores where all cores share a common interface to main memory and groups of cores are subject to a common clock frequency.

This work presents an approach to apply previous knowledge about the energetic characteristics of microprocessors to computer systems based on a multi-core design. By employing an energy-aware scheduling algorithm, which collects specific energy-related information regarding the processes in the system, the operating system can determine the most efficient performance state the respective processor should operate at. This concept is augmented to the multi-core case by coordinating the set of individual performance state requests according to processor topology and the policy objectives the user can specify. The side effects shared memory bandwidth has on our policy are overcome by employing a heuristic that models the memory-related behavior of our platform.

We implemented energy-efficient scheduling for the Linux operating system kernel. Our evaluations show that the proposed design allows to run a multi-core system at optimum efficiency under varying workload conditions. The significant gains in energy efficiency that are possible with some workloads come at no additional hardware requirements or costs.

Acknowledgements

I would like to thank my advisors at the University of Karlsruhe, Prof. Dr. Frank Bellosa and Andreas Merkel for their constant support and the opportunity to conduct research on such a rewarding topic as power management with regard to the architecture of computer systems is. Equally, I would like to extend my sincere thanks to Prof. Dr. Gernot Heiser and his ERTOS group at the NICTA research lab in Sydney, Australia. Without the most inspiring research environment and the interaction with his team this work would not have been possible in this form. Here, especially, I want to mention Dr. Peter Chubb, my supervisor, for his input and guidance as well as David Snowdon and Etienne Le Sueur for the fruitful discussions I had regarding my thesis work.
Finally, I want to thank my parents and family for the trust and love I receive and their selfless dedication to support all of my goals during the time of my studies.

Contents

Chapter 1

Introduction

1.1 Improving Energy Efficiency

With the advancements in semiconductor technology, integrating a steadily increasing number of transistors with smaller feature sizes on the same processor die area, power dissipation is becoming a serious concern for the design of computer systems. As an inherent property of processors based on a Complementary Metal-Oxide Semiconductor (CMOS) design, power consumption of an integrated circuit is a function of clock frequency and supply voltage. Recent processor microarchitectures offer the capability to adjust these hardware parameters during operation in order to allow for a trade-off between power consumption and performance. While today's operating systems often use these mechanisms to reduce hardware power dissipation based on the current load situation, these measures only have a limited benefit on system efficiency. By collecting specific energy-related information about the software workload, it is possible to selectively apply frequency- and voltage-scaling during phases where processor performance does not depend as much on clock frequency and consequently results in an increase in energy efficiency.

Previous research in this area demonstrated that it is possible to dissipate less energy for the same amount of work by means of an improved – i.e., *energy-aware* – scheduling strategy inside the operating system kernel. In this design, necessary information about the exact composition of the currently executed workload is obtained by means of processor hardware registers counting performance-related events. Together with dynamic frequency scaling, these commonly available architectural features can be exploited to render processor operation more efficient. For the end-user, energy efficiency translates into increased battery life, a lower energy budget, and more dependable operation due to optimized power consumption – without involving additional hardware requirements or costs.

1.2 Chip Multi-Processors

The microarchitectural design of processors recently made a significant leap with the emergence of multi-core processor architectures featuring multiple CPU cores in a single physical package. Processor architects try to cope with the obstacles of increasing design complexity by co-locating a number of identical processor cores on a single silicon die. For economical reasons, such a design usually implies that some hardware components have to be shared among the individual cores. Most prominently for chip multi-processors, this is the common interface to main memory and basic circuitry regarding clock signal and supply voltage. With frequency- and voltage-scaling as primary means to realize an energy-efficient operating system policy, such an architecture creates mutual dependencies that have to be considered when optimal decisions in terms of energy dissipation are necessary.

While existing proposals for energy-aware scheduling policies are applicable to a variety of systems based on single-core processors, ranging from embedded systems to more common desktop CPUs, they are oblivious to the apparent peculiarities of chip multi-processors. Disregarding the particular topology of the underlying hardware will most certainly result in a suboptimal usage of processor performance states or even an increased energy dissipation. Obviously, power management techniques have to keep up with the advances of processor architecture. In fact, considering the range and number of deployed systems based on a multi-core design today, optimizing energy consumption by means of a minor modification to the operating system kernel may have a sustainable effect. However, the challenge associated with systems based on a multi-core design has not been sufficiently tackled yet.

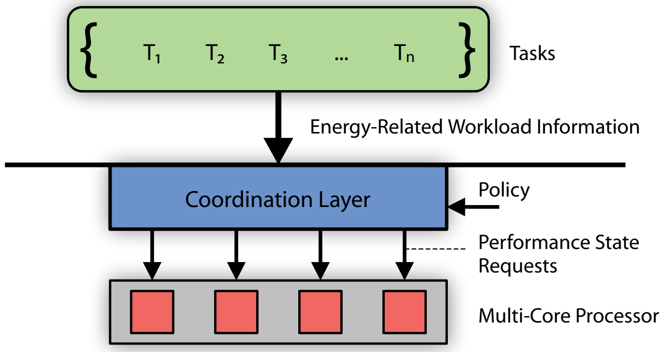
1.3 Energy-Efficient Scheduling for Multi-Core Processors

The objective of this thesis is to modify an operating system scheduler in a way that hardware, based on a multi-core design, runs at optimal efficiency while system workload and utilization may vary. The best choice regarding processor frequency and voltage at a certain point in time can be determined a priori (i.e., offline) or at runtime, but depends inherently on the type of instructions that are being executed. Chip multi-processors complicate this situation as now multiple streams of instructions are being processed in parallel while, at the same time, changes in clock frequency may affect not only one core of the processor package.

An approach to energy-aware scheduling by collecting information related to the behavior of each task in the system has distinct advantages. Provided that these *task profiles* contain information specific to the characteristics of a process, at each point in time when a scheduling decision is due, the operating system scheduler has exact knowledge about each task's properties and can determine optimized decisions regarding a multitude of objectives. The performance monitoring facilities all major processor architectures feature today prove to be suitable indicators for energy-related properties of the currently executed workload. As application behavior typically depends on input parameters at runtime, compared to offline analysis, this reactive *best-effort* approach is well suited for practical computing scenarios. Moreover, such a design can be realized entirely in software and does not require special preparations prior to deployment. We argue that an advanced operating system should exploit all available mechanisms provided by modern processor architectures in order to support performance of the underlying hardware at optimum efficiency.

Since the individual cores of a chip multi-processor are still based on the same design as previous single-core processor architectures, the basic principles for leveraging more efficient CPU operation are likely to remain unchanged. Assuming the context of a single processor core, a former decision to scale CPU voltage and frequency may still prove valid. However, given the possibility of hardware dependencies, a universally applicable energy-aware solution has to ensure that the employed strategy honors policy objectives to a similar degree for multi-core processors.

Supported by an online energy estimation design, our solution exploits the concept of per-task energy profiles for the implementation of CPU power management on multi-core systems. Based on these profiles, carrying information about the rate of specific types of energy-related events, it is possible to determine the impact scaled frequency has on power consumption as well as performance. By employing a simple model based on two heuristic algorithms, it is possible to determine the best performance state on a per-core basis. As this decision has to remain consistent in a multi-core context, these transition requests will be reevaluated by an additional software layer taking possible hardware dependencies into account. This basic design principle is depicted in the following diagram.


T_1 T_2 T_3 ... T_n
Tasks
Energy-Related Workload Information
Coordination Layer
Policy
Performance State Requests
Multi-Core Processor

Based on an existing energy infrastructure allowing task-specific energy estimation, we implemented an energy-efficient scheduling policy for the Linux operating system kernel. We augmented this design to support a new processor microarchitecture and extend the concept of energy-efficient scheduling to the multi-core domain. Our evaluations show that significant improvements in energy efficiency are possible with hardware systems based on a chip multi-processor architecture.

Thesis Scope

Throughout this thesis, we will focus on operating system tasks that are *not interactive* (i.e., utilizing the processor hardware for extended periods of time), as these processes offer optimal behavior to save significant amounts of energy. For interactive tasks, it is likely that the effort incurred by transitioning to another frequency setting for only a short period of time might outweigh the benefit of an adjustment in the first place. Further, in order to limit the complexity of analyzing the characteristics of a multi-core processor with all possible combinations of frequencies and tasks, we constrain our analysis to the *utilized* case. Meaning, there are only as many tasks active in the system as there are processor cores. Regarding a particular point in time, a situation with more tasks is a special case of the task configurations we will analyze.

1.4 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 elucidates the background for CPU power management and the architecture of multi-core processors in particular. The second part of the chapter presents related work in the field of energy-aware scheduling. Chapter 3 analyzes and verifies the energetic characteristics of our hardware platform with regard to previous results and develops a solution that augments our policy design to chip multi-processors in general. Chapter 4 discusses the issues relevant for the actual implementation of energy-efficient scheduling for the Linux operating system kernel, while Chapter 5 evaluates and confirms the soundness of our design. Finally, Chapter 6 concludes this work and puts it into the context of possible future research.

Chapter 2

Background and Related Work

The purpose of this thesis is to apply and augment existing power management policies to multi-core processors. This objective combines two fields of recent attention: the need for energy-efficient operation of computer hardware becoming more and more crucial and software having to deal with the burden of complex processor architectures featuring a steadily increasing number of cores on a single chip. Knowledge about the energetic characteristics of microprocessors in general is a prerequisite for an adequate analysis of situations where energy actually can be saved, as is the understanding of the implications imposed by the special architecture of chip multi-processors.

We will provide this background information in the following sections of this chapter. Subsequently, we will present the results of previous research that relates to the main themes we have to take into account for our design. As power management is a wide field, we will identify clues among this work that may as well apply to our target platform, serving as foundation towards an energy-aware power management policy for multi-core processors.

2.1 Background

2.1.1 Processor Architecture

Since the emergence of the first highly integrated circuits, information technology is driven by the advancements that have been made with the fabrication of semiconductors. The past decades have seen a constant increase in processing power and a decrease in feature size (which is usually determined by the width of a single transistor's gate). Smaller structures means considerably more space for transistors, which allows for more elaborate functional units to be integrated on a chip of the same size. While techniques like *out-of-order execution* and *multiple issue* found their way into more advanced generations of microprocessors, performance does not necessarily scale linearly with the additional costs in resources. Many of these enhancements depend on the predictability of the workload (i.e., the stream of instructions the CPU has to process) or have to take dependencies into account, which might arise between operands. Obviously, a microprocessor spends a considerable amount of logic and power to speculatively calculate results, which might not even contribute to the final outcome of a series of instructions.

A speculative design is inherently limited by the fact that there is no guarantee for a constant rate of correct predictions, despite the presence of multiple potentially underutilized functional blocks. In terms of power consumption and efficiency, these diminishing gains do not justify the accompanying loss in die space. Therefore, the implementation of resources for ILP purposes seems to be economical only up to a certain level. Superscalar processors exhibit an architecture-specific instruction issue capability per clock cycle. Taking phases of narrow issue utilization into account, other ways of harnessing the computational resources effectively had to be found [13]. In conjunction with the continuous increase in processing speed outpacing memory access latencies, both developments are sometimes referred to as the *ILP- and memory wall*, which pose the two major obstacles to the recent development of more advanced processor architectures.

Typically, the software that runs on a microprocessor shifted towards multi-threaded workloads in the last years. From a hardware point of view, different independent streams of instructions offer the opportunity to execute threads in an overlapping fashion, even if there is only one processor core worth of functional blocks.

The frequency headroom, which is made possible by continuous advances in semiconductor technology, could not be translated into higher clock rates lately due to the challenge to keep pipeline depth low and avoid undue critical path delays. However, provided that an improved fabrication process also leads to more space available for additional logic, new approaches to satisfy the need for more processing power had to be conceived. Maintaining the focus on threaded workloads, the next step in this evolution is to not only optimize the utilization of parts of a single processor but to provide multiple equally potent instances of the same core – all fitted with their own exclusive ILP optimizations as well as dedicated cache memory. Depending on various design choices all or a subset of cores may be co-located and share the same silicon die. Either way the interface of the physical package appears as a single processor, although in reality there well may be a *multi-chip module* inside.

From a hard- and software point of view, these *Chip Multi-Processors* (CMP) represent a special form of symmetrical multiprocessing with a number of equal processors (in this case processor cores) of the same microarchitecture and ISA. Within certain boundaries and depending on the workload (embarrassingly parallel applications are a prominent example), having more than one core available can lead to a multiplication of exploitable processing power on the same die area. However, software support is required since this form of parallelism is not dynamic and depends on prior decomposition of the workload so the hardware can assign execution threads to the separated resources – unlike multiple issue, which takes place during runtime. Thus, CMP represents a natural approach to cluster resources in the era of billion transistor chips [9] and at the same time helps to alleviate potential problems at the design level regarding increasing interconnect delays [12]. Apart from an independent core layout, CMP chips also share several key components, e.g. the last level of the cache hierarchy or even the front side bus, an interconnect that interfaces the entire package to the system's main memory subsystem.

Shared resources are likely to pose a major system bottleneck and may incur interference among otherwise independently acting processor cores. This is the reason why the particular architectural organization of a processor may not be neglected with regard to both performance and efficiency. Hence, for the task of designing a valid solution, we will have to pay special attention to the issues architectural constraints may imply. First, however, we will clearly state what the notion of efficiency means in the particular case of microprocessors and how this metric relates to power and performance.

2.1.2 Power and Performance

Integrated circuits, especially chips based on a *Complementary Metal-Oxide Semiconductor* (CMOS) design, which is the predominant technology used for microprocessors nowadays, dissipate power due to several distinct physical effects. It is important to note the reasons why power dissipation turns into a problem with the evolution of modern microprocessors and which parameters may have an influence on the energetic profile of CMOS circuits in general.

Recent generations of CPUs exhibit an extremely high complexity, turning the task of working with or even calculate a precise power model into a hard problem. Therefore, to be able to estimate the power requirements a certain mode of operation will have, in most cases it is necessary to abstract the model to a basic level that still reflects the dominant effects observed on the outside.

In the domain of electrical circuitry, power is one of the major quantities, since it has an impact, for example, on thermal design and also the efficiency a device can provide. In an electrical context, power is typically dependent on the actual supply voltage and the amount of current running through the device under test. Power is measured in units of *Watt* and is closely related to energy as power denotes *the instantaneous energy at a certain point in time.* Basically, for each operation a microprocessor performs the current that runs through the IC for this certain amount of time gets converted to heat. During the remaining time static leakage effects usually dominate.

In theory, CMOS logic should only dissipate minimal power due to the design principle where either the N-MOS part of the logic or the complementary P-MOS part gates. Despite the fact that there should be no current path from supply voltage (V_{dd}) to the ground (V_{ss}) whatsoever, charging load capacitances involved with each field-effect transistor makes up a major part of the overall power consumption. Since the effect increases with the rate transistors are switching between states, this term of the equation becomes the dynamic part (or *dynamic power*), which depends on the frequency the circuit runs with. The reason for an idle microprocessor to consume a fixed amount of energy per time is due to leakage effects within the transistor structures (gate as well as source/drain leakage) and even between the interconnects, which become an important concern as feature sizes shrink. In fact, today's microprocessors dissipate an increasing fraction of their energy on leakage [24]. Since feature sizes decrease to a level where the dimensions of the gate's dielectric are reduced to a height of only several atomic layers, these effects also contribute to power consumption on a macroscopic scale. Without any advances in substrate technology, it even becomes mandatory to add supplementary logic for the processor to be able to cut unneeded parts of the logic from the power plane in order to save static power for this area of the die.

In the context of power analyses, logic designers and system builders often resort to a principal equation similar to (2.1), which reflects the relationship between power dissipation and the main parameters driving an integrated circuit [21]. With increasing power levels, the associated development of heat cannot be neglected. This is the reason why the TDP (thermal design power) has also become a growing design constraint.

$$P \sim CV^2f + VI_{leak} \tag{2.1}$$

As previously discussed, the given equation covers two main effects, namely the dynamic and the static components of a CMOS circuit's power dissipation. Provided that some activity, which invokes an increase in dynamic power consumption, is taking place on the chip, it is possible to reduce the chip's dissipation significantly by either reducing the clock rate, which usually comes at a certain performance penalty, or even to a more effective degree by reducing supply voltage. This is because V contributes quadratically to the first term compared to the linear relation in power when changing f (i.e., the clock rate). Although this seems to offer a tempting opportunity to just lower the voltage at runtime, this cannot be done arbitrarily since a certain voltage level is required when the microprocessor is supposed to run reliably at a high frequency.

Power has already become an important architectural constraint in today's design of microprocessors and will most certainly remain a serious problem in the future taking the constant trend towards increasing integration sizes and the need for high density computing environments into account. Considering a large deployment of computer systems, either in a high-performance computing environment or simply on a global scale, any improvement in energy efficiency may pay off significantly.

On the software side, throughput, apart from latency, is one of the most important factors for characterizing the performance of a computer system. In computer science the term performance describes an amount of operations that can be processed by a system in a given period of time. Performance is closely related to the term energy, since a benchmark application (i.e., a well defined sequence of instructions) takes a certain period of time to complete and therefore dissipates a specific amount of energy. For a hypothetical non-pipelined in-order microprocessor this would be the sum of the energies spent for each operation in that sequence. Having a configuration doing the same amount of work at a lower power level in comparable time would mean an increase in *efficiency*.

2.1.3 Power-Related Hardware Mechanisms

Compared to previous generations, recent CPU architectures increasingly employ supplementary features, for example access to functionality that is related to the microprocessor's external interface or some integrated special-purpose circuitry. Among these components are the local APIC, thermal monitoring capabilities (TM2), architectural performance monitoring, as well as control over core frequency and processor supply voltage [8]. All these circuits are likely to be processor family or even chipset-specific and are therefore made accessible via a range of special control registers rather than by means of an additional set of instructions for each new type of extension. Consequently, the organization of this model-specific address space of registers may vary significantly among CPUs of different vendors. *Model-Specific Registers* (MSRs) can be uniformly accessed by reading or writing predefined values via `rdmsr` and `wrmsr` instructions from a valid register address. Reading from an MSR allows for system software to determine the current state of this particular part of the hardware. This essentially means, while the software keeps executing against a consistent and immutable ISA, software also becomes privileged to change crucial external parameters, which have an immediate influence on the operating point and performance of the processor, but not the general execution of instructions apart from that.

Obviously, both the respective microprocessor and its chipset have to provide support for the implementation of a dynamic change of core voltage and frequency at runtime. With the evolution of processors and mainboard chipsets across desktop, server, and mobile computers, virtually all recent platforms support a proprietary implementation of dynamic voltage and frequency scaling (DVFS). Depending on the architecture of a particular platform, however, there are different clock signals, which may or may not have requirements regarding a fixed frequency ratio. Intel's processor architecture typically uses a front side bus to interface to the system's northbridge and main memory. Since the core clock gets derived from the FSB by a PLL (phase-locked loop) multiplier, frequency can typically be increased in fixed multiples of the bus frequency while peripherals are bound to be clocked according to the bus system. However, changing the FSB to core clock ratio usually comes at a latency penalty during which the PLL settles to the new frequency.

Fortunately, new generations of microprocessors profit from recent efforts in the industry on optimizing hardware power management mechanisms, resulting in significantly improved performance state transition latencies [7, p. 17]. This in turn encourages performance state adjustments to be issued at high frequencies, enabling system software to implement more elaborate CPU power management policies.

2.1.4 Power Management

The notion of power management describes the capability to control *when* and *how* certain activities in the system may take place. Compared to a conventional computer system, such fine-grained control translates into the ability to use the present hardware in a more energy efficient manner during phases of varying usage patterns.

With the introduction of ACPI [6], the *Advanced Configuration and Power Interface*, the complexity associated with properties of a certain platform in terms of power management – meaning the management of hardware details – gets hidden. ACPI is an interface specification that allows the system to communicate abstract information about available power states to the operating system. These states can be applied to individual devices as well as the entire system, which is under the control of a power management policy. The advantage of this approach is that the operating system gains comprehensive control over all available power saving mechanisms while not interfering with a device's firmware-based mechanisms.

Since the main CPU is of an essential role for system operation and performance, this component is represented by an own class of ACPI devices. Recent processors can enter different CPU and power states, which get mapped to a number of available power saving modes and frequency states the platform exposes. Generally, there is a distinction between CPU power states (C-states), performance states (P-states) and throttling modes (T-states) [6].

Valid performance states for a certain family of processors are usually determined by the manufacturer who designed and tested the hardware for compatibility. P-states have an influence on the operation point of the processor as they determine at which voltage and frequency the circuit runs. According to the ACPI specification, valid P-states are indexed by an integer number in ascending order beginning at zero. As a result, P_0 denotes the state of nominal processor speed and all lower frequency settings are organized as P-states with a higher index number. We will abide by this notation for the rest of this thesis.

Transitions between power states (C-states) occur when the system enters or exits an idle situation where no operations at all are pending to be executed, at least until the next interrupt event. A lower power state allows the processor to disable unneeded parts of the logic in order to conserve energy while pausing. Low power modes often come at a higher cost in terms of latency, whereas a simple `hlt` instruction (equivalent to ACPI C_1 state) incurs the smallest delay to get the processor to full operation mode again. Since the processor is able to almost instantly resume its work, there is not that much opportunity to shut down functional units than compared to the deeper sleep states.

Hardware traits like multi-core organization and simultaneous multi-threading capabilities cause dependencies among available hardware states and therefore introduce another dimension of complexity to the coordination of processor-class devices. Power management policies must not neglect such platform peculiarities since this most certainly will result in suboptimal energetic behavior. Given a compliant ACPI firmware implementation, supplementary information about hardware dependencies is usually provided to the operating system by means of special purpose ACPI tables [6, Ch. 8.4]. Accordingly, system software can take such hints into careful consideration in order to implement a policy that actually agrees with the effective state the hardware operates at.

2.2 Related Work

Information regarding the power and energy behavior of a system can be obtained in several ways. There are two main approaches to *energy estimation*. Offline analysis can be conducted by simulating the behavior of a particular system along with the specific software that is subject to investigation. This approach can yield detailed information about energy consumption based on a mathematical model. The advantage of such a model is the possibility of increased simulation accuracy. On the other hand, keeping the model at a basic level may allow saving computational resources and time (Wattch simulator by Brooks et al. [3]). Another offline method for energy estimation is to collect data by means of a *Data Acquisition Device* (DAQ). By sampling real world data, this method can be superior to simulation, provided that the needed accuracy of the involved measurement setup is given. A shortcoming of all methodologies based on an offline approach is the requirement of prior knowledge of the exact workload that is to be run on the deployed system [17]. For *real-time* systems this requirement is likely to be met, since these systems are required to be analyzed prior to their actual deployment anyway. However, offline approaches allow for optimal power management decisions to be made regarding the entire time of execution. Given that conditions remain unchanged, this obviously also applies to all future runs.

Nevertheless, in almost all cases software behavior is dependent to various input parameters and thus cannot be predicted *a priori*, rendering the offline approach impractical for commodity (i.e., non-realtime) systems. The complement approach, on the other hand, acknowledges this fact by striving to base decisions on actual observations at runtime. In either approach, energy-related behavior is supposed to be the basis of any power management decision. This is the reason why *online energy estimation* tries to achieve some kind of prediction based on present and past behavior. This prediction obviously does not guarantee for optimal decisions and rather pursues a *best-effort* policy approach. Nevertheless, to warrant any such best-effort claim, as precise knowledge as possible about the tasks and the system itself is necessary. Since an online energy estimation approach seems to be practical enough to be actually implemented in future versions of energy-aware operating systems without any additional hardware requirements, the increasing demand in energy-efficient computer systems well justifies the effort in implementing such a mechanism.

2.2.1 Event-Driven Energy Estimation

Although performance counting facilities of microprocessors have been used to gain performance related information before, Bellosa [1] pioneered in proposing to take direct influence on scheduling decisions by means of such runtime information. The work unfolds a basic relation between energy consumption and accountable performance events (i.e., activity of distinct hardware units) for its test systems. By performing a calibration for a particular system offline, it is possible to turn selective event information into runtime energy information. Meaningful performance events related to *integer and control flow instructions*, *floating point instructions*, *L2 cache references*, and *memory accesses* are identified as being relevant to energy consumption. Further, it is reasoned that proper power management in an operating system can only take place if task-specific energy accounting is possible. By instrumenting the operating system kernel accordingly, this energy-related information can be associated to a task's runtime context. In fact, previous scheduling code has been entirely *CPU-centric*, whereas in-depth information about hardware activity enables the scheduler to tune its decisions for better overall performance (e.g., cache thrashing situations the scheduling code has been oblivious to in the past) or towards specialized energy-related objectives. The paper even envisions the possibility of adjusting processor frequency for each task individually to operate at an energy-optimal level at any time. Furthermore, any energy-aware scheduling policy will be mostly transparent to user-level applications since necessary changes will happen while the kernel is executing privileged code.

Before proceeding, however, it has to be noted that the presented methods, as the name suggests, perform an *estimation* of dissipated energy. This obviously involves an inherent margin of error as the instrumented performance counter events are geared towards giving meaningful hints regarding performance rather than energy dissipation. For example some chip activities cannot be accounted for since no proper performance event is available that covers just that specific case. As previous related work [1, 25] already indicated, a linear approach by counting a set of events as weights towards a momentary energy value proves useful, yielding the linear combination of (2.2).

$$E = \sum_{i=1}^{n} a_i \cdot c_i \tag{2.2}$$

Here, a_i is the predetermined *energy weight* and c_i the event count during the past estimation interval. Kellner [14] worked on the first sophisticated energy model by manually selecting adequate performance events for the Intel Pentium 4 generation of microprocessors and reports estimation errors of up to 10% for some applications (29% for the worst case that could be identified). Nevertheless, employing performance monitoring counters is a suitable means to estimate power dissipation with no additional instrumentation necessary and incurring only minor performance overhead. Further, due to the high rate of events, it is possible to estimate energies for relatively short timespans as well. Particularly brief periods of application I/O, however, are not necessarily representative in their composition of instructions compared to the rest of the code.

By proposing a methodology for online event-driven energy accounting, this paper has laid a solid foundation for subsequent work aiming to further improve energy-aware operating system policies. The event-driven approach has proven to be superior to previous approaches only taking idle phases versus periods of utilization into account [11]. Assuming unpredictable application behavior, recurrent analysis of performance and energy-related hardware events allows for the operating system's measures to adept in order to implement its energy-aware policy.

Similar to this approach, the solution of this thesis will use the methodology of energy estimation to realize more energy-efficient system behavior. Although not aiming at building a sophisticated energy model from performance event data, our design will use and extend these mechanisms later to collect and evaluate energy-related events.

2.2.2 Process Cruise Control

The proposal of *Process Cruise Control* by Weissel and Bellosa [25] as a novel operating system policy is based on previous achievements of event-driven energy estimation, which encourage the instrumentation of *Performance Monitoring Counter* (PMC) hardware in the processor to have a beneficial influence on scheduling decisions – i.e., an energy-aware policy. In contrast to earlier work, the goal of this policy is not to constrain overall energy consumption, but to yield gains in energy efficiency (see Section 2.1.2). In order to find clues relevant to this goal, the paper attempts to identify components which either benefit from clock scaling or not and the types of tasks that are using these components. As already anticipated by Bellosa [1, Sect. 4.2], the authors discover a relationship where a high count in memory references and a low number of instructions indicates that the speed of execution is dominated by memory latency. Throttling the processor's clock speed on behalf of such a *latency-bound* application should therefore end up in only a minor performance penalty. Apart from that, the work defines an additional constraint where performance degradation due to frequency scaling should not exceed 10% at any time.

To accommodate the fundamental relationship between performance penalty due to instructions, which scale with core frequency, and opportunities for energy savings determined by latency-bound workloads, the authors define the concept of application-dependent *optimal processor speeds*. This setting, by definition, involves the minimum energy profile for a given workload. Determining an optimal frequency parameter under a given performance constraint is subject to a benchmarking process that is platform-specific (and so is the resulting data). However, in order to make this information accessible to the operating system's scheduler without incurring undue performance overhead, they approach this problem by partitioning the parameterized space into *frequency domains*. In order to find the optimal execution speed for the next timeslice, two input parameters to this model are determined by performance counting facilities – namely, the number of instructions and memory requests per clock cycle. Hence, the decision-making process can be broken down to a simple table lookup for the respective energy-aware algorithm.

By choosing an interval-based approach, this work avoids any concern regarding sophisticated phase detection techniques, which have been subject to extensive research in the past. Since scaling decisions are met recurringly for each new time interval (i.e., the next scheduling timeslice), this strategy also provides a solution to program phase detection. Clearly, the possible resolution of phase changes will be limited. Employing a more sophisticated detection technique, the accuracy (and thus possible gains) could be improved. However, any effort made in this direction will lead to a trade-off between additional overhead and diminishing gains in efficiency.

The work of Weissel and Bellosa manages to enhance the operating system's scheduling decisions by a new dimension, which can be of great benefit especially in environments where energy is a limited resource (e.g., battery-driven devices or high density computing). By using energy economically, significant savings are achievable; up to 22% of the dynamic energy for an embedded platform at 10% performance loss.

Towards a viable power management policy ready to be deployed with general-purpose computer systems, the presented online-feedback approach qualifies by involving only moderate effort on the software side, whereas system requirements are easily met with newer generations of computer hardware. In our design, we will use the same basic approach in order to implement an energy-efficient scheduling policy for the new platform. Moreover, we will evaluate whether it is feasible to enforce a timing constraint similar to the proposal of Bellosa [1].

2.2.3 Workload Classes

Similar observations regarding the potential to save significant amounts of energy have been made in related research papers as well. The basic idea is to split contribution to performance loss between major classes of operations instead of using an application's IPC rate as indicator for performance, which turns out to be an undue simplification [15]. It is necessary to distinguish between processor activities, which are dependent from core frequency and events that are latency bound and therefore subject to separate external clocking signals. Differently put, we have to distinguish between *on-* and *off-chip* [5] activity, meaning, CPU- and memory-intensive phases. The respective authors denote this approach as *workload decomposition* [4] or *partitioning*. Kotla et al. [15, Sect. 3.1] investigate the behavior of the target platform with a synthetic benchmark program to tune between memory- and CPU-boundedness, respectively. Whereas on-chip activity still scales while changing frequency settings, the effect of memory saturation becomes apparent when regarding throughput at different boundedness ratios. For their studies concerning the impact of varying core frequencies on different levels of memory intensity, the authors employed fetch-throttling. They argue that adequately throttled processor cores can mimic the energetic and memory-access behavior caused by frequency scaling. By developing a simple model to predict application performance at a lower frequency it is possible to determine the optimum clock parameter depending on its memory intensity.

Our design resorts to the given notion of workload classes. It resumes and consequently exploits the approach this paper proposed and studied only preliminarily. In contrast to a sophisticated energy model, which seeks to assign a large number of functional units and performance events with respective energy weights, the workload class approach reduces the energetic analysis of arbitrary application workloads to a meaningful yet simple model. Having a model as concise as possible keeps additional overhead incurred by the maintenance of an energy-aware scheduling policy at a minimum. We will have to identify a mechanism, which is capable of carrying and maintaining this abstract workload class data on a per-task basis.

2.2.4 Energy-Aware Policies

Closely related to these aspects of energy estimation is the development of scheduling strategies that are able to implement thermal policies. Kellner [14] also did initial research in *Dynamic Thermal Management* by deriving a CPU thermal model from data yielded by energy estimation (see Section 2.2.1). Without the need for online temperature measurements, the author demonstrates the possibility of throttling the execution of tasks according to their energy-specific characteristics and the thermal requirements of the system. As a result, any given temperature limit will not be exceeded.

Given that the workload is sufficiently diverse in its energy profile, a multi-processor system opens new levels of freedom in terms of thermal management. Merkel [18] studies this case in order to find more optimal methods of balancing temperature among processors. Caused by emergency circuitry in today's processor architectures, critical device temperatures trigger a throttling mechanism, which can incur undue performance penalties, while other processors might be underutilized and therefore still operate well within temperature limits. The combination of energy-aware policies – *energy balancing* and *hot-task migration* – proposed in this work effectively mitigates thermal imbalances among processors. Therefore, in the event of aggravated cooling conditions, the operating system's awareness of a task's energy profile can significantly help in improving system throughput. In order to accommodate the Linux scheduler in its energy-aware decisions, Merkel puts an energy infrastructure in place, consisting of *Energy Estimator*, *Energy Profiler*, and an *Energy Aware-Scheduler*. The first component provides the means to evaluate the processor's performance counter MSRs and keeps track of the number of events that occurred since the last invocation. To turn this data into per-task values – i.e., counting the events that happened during the execution of that particular task –, the energy profiler invokes an evaluation function on each context switch. The energy profiler in turn is responsible for assigning energy values to the respective task's energy profile as well as maintaining an average power consumption rate for each processor. The consumption rate has been adjusted to mirror the thermal behavior of the respective CPU by means of an exponential average function [18, Ch. 3.5.6].

This energy framework provides a starting point for a set of extensions we will apply in order to support a new processor platform and accommodate the type of task-specific workload information our design requires. We will delve further into the details of the energy infrastructure while we elaborate on our solution in the subsequent design and implementation chapters.

2.2.5 Multi-Programmed Workloads

As our previous reasoning about the particular architecture of multi-core processors already suggested (see Section 2.1.1), we will have to consider any special case that might arise when a system bottleneck starts to impose performance penalties on concurrently running tasks. Computer systems inherently exhibit upper-bounds in their processing capabilities: for instance, operations per time or in case of the memory subsystem read-write accesses per time. Side effects can heavily impede performance if vital system devices are shared between multiple components. Since efficiency is closely related to performance, we have to identify and analyze these cases energetically in order to work out the optimal management policy to handle such kind of events in an efficient manner.

Similar issues regarding degraded efficiency due to memory bus contention in SMP systems have been approached in the past. Compared to the processing capabilities of a recent microprocessor, the speed of memory lags behind significantly; also commonly known as the memory wall [2]. This general situation gets aggravated if even more devices, as in a multi- or chip multi-processor system, contend for access to caches and memory as typical shared resources. Koukis and Koziris [16] did related research in order to avoid saturation of system bottlenecks. They chose to instrument performance counters to gain information about memory access patterns. Accordingly, they propose a policy of *bandwidth-aware scheduling*, which basically describes an optimized co-scheduling algorithm assisted by heuristics based on task bandwidth utilization. The authors report significant gains in throughput (40–48%).

With the focus on energy management considerations for multi-programmed Intel Pentium 4 processors (comparing SMT and CMP systems of similar microarchitectures), Winkelmeyer [26] reports analogous observations regarding shared resource effects. By running several synthetic benchmark applications in parallel, the author tries to identify the implications these combinations have on power and performance. For the given architecture, the number of micro-operations retired is an indicator for application performance while the number of bogus micro-operations (i.e., a value based on a speculative misprediction) is an indicator for power dissipation. Apart from the predominantly predictable behavior of a CMP system across a broad range of CPU-bound applications (since functional units are exclusively partitioned), for pairs of memory-intensive workloads notable effects of interference emerge. The author reasons that with the given behavior, it is possible to limit energy consumption of tasks by scheduling them according to their estimated energy profile and how this could be assisted by using processor performance states.

2.2.6 Multi-Core Caches

Efficiency benefits by leveraging processor caches in an optimal way to improve thread communication and minimize cache contention is beyond the scope of this thesis. We refer to related work, showing that migrating dependent groups of tasks to processor cores sharing a part of the cache hierarchy can further improve efficiency [23].

2.2.7 Energy-Efficient Scheduling

Given that the workload situation warrants any freedom in co-scheduling a set of tasks – viz., there are more tasks eligible to run than processors available – an advanced scheduling algorithm may exploit additional information in order to even out or minimize the interference among tasks regarding shared memory bandwidth (s.a.) and cache utilization [2, 10]. As already stated in the introduction, we aim to focus our attention on the utilized case. We point out that situations of over-utilization can be *temporally decomposed* into a configuration of n concurrently running tasks (with n = number of cores) during a particular window in time. Accordingly, we will approach limited memory bandwidth from the perspective of a given static configuration of tasks. Still, any resulting conclusion of our work will also be valid for the special case, contributing to the overall understanding of multi-core characteristics and the improvement of advanced scheduling policies.

2.3 Minutiae

Since many of the notions we will use in the following chapters have an association to more than one physical or logical parameter, the usage of certain common terms might become ambiguous. For the sake of clarity, we will define these notions and their usage beforehand.

According to Section 2.1.4, P-states are indexed in ascending order with decreasing frequency. Hence, we speak of a high or higher P-state when a setting at a lower clock frequency is meant. However, the terms scaling up or down are usually understood with regard to frequency, meaning we scale up to a higher and down to a lower clock frequency.

In the next chapter, we will use performance as another parameter to make determinations regarding efficiency. Typically, we normalize values denoting performance to a default value of 1.0 for the speed a workload is running at the highest processor frequency. Accordingly, if we speak of slowdown, this term is usually understood in a context of performance and throughput, resulting in factors of less than 1. A slowdown factor of $\sigma = 0.8$ describes an increase in runtime of $25\% = 0.8^{-1} - 1$. Thus, if we speak of increased runtime or a slowdown in terms of time, this value is usually provided in percent or identifiable as a factor greater than 1.

Chapter 3

Design

3.1 Problem Statement

Referring to the motivation of this work in the introduction, the primary challenge with realizing an energy-efficient scheduling policy is to find regularities and rules in the energetic behavior of the system subject to analysis. The evaluation of related work showed that more efficient processor operation is feasible for a range of systems and yielded valuable clues how such policies can be implemented. Towards a valid design for our particular hardware platform, we first have to verify the applicability of these clues by analyzing the system energetically. Based on this analysis, we can develop an adequate strategy for our policy to adjust processor frequency and voltage to the most efficient state, taking the energetic characteristics of the current workload into account. Mutual influences between cores have to be regarded by this analysis as well in order to achieve optimal results and ensure the universality of our solution. Due to the large number of possible configurations with tasks and performance states in the multi-core case, the overall approach has to be methodical.

3.2 Design Overview

Throughout this chapter, we will elaborate on a viable design to allow for energy-efficient operating system policies to be implemented in a generic manner. Following the approach of previous work (see Section 2.2.2), we will use energy-related data specific to a particular task to determine the performance state, which is optimal while this task is running on a processor. By employing an online energy estimation approach, our solution is independent from offline application analysis and can adapt to varying workload patterns at runtime. It makes this design suitable to be deployed as part of a general purpose operating system.

The starting point of our design proposal is the distinction between multiple classes of processor events (*on-chip* and *off-chip*) in order to support energy-related classification of the current workload. The generic infrastructure is complemented by a simple heuristic, which will determine the most efficient performance state the task should run with. In general, the prediction of future application activity based on past behavior does not allow for accurate decisions or guarantees. However, following a *best-effort* approach, employing a heuristic as mechanism still promises to yield considerable gains in energy for memory-intensive workloads. With the parameters of the policy tuned correctly, the respective processor may run at an efficient state most of the time. We augment the proposed policy to the *multi-core* case by coordinating incoming performance state requests from each core in a way that takes physical dependencies into account and transitions the overall processor package into the most beneficial state.

Another important constraint, which adds to the complexity of the situation, is the processor's shared interface to external memory. Especially since memory is the same resource that allows for energy savings in the first place. Available bandwidth is inherently limited, meaning once multiple cores perform accesses to memory, they will only obtain a share of this resource. Since ultimately a task has to wait for a memory operation to complete, insufficient bandwidth can turn an otherwise compute-intensive task into a memory-bound workload. Obviously, this condition has an important influence on our policy. We take this architectural constraint of chip-multi processors into consideration by developing an appropriate bandwidth model, which provides the input data for the proposed threshold heuristic.

3.3 General Reasoning

In the introductory chapter and Section 2.2, we already mentioned some details regarding the goals and the approach this work will take. Before starting to reason about any specific decision in design, along with the accompanying advantages and shortcomings, it is necessary to define our objectives precisely. Thus, we will declare the principal guidelines for our policy first: that is, the goals we strive to accomplish and any foreseeable behavior we might have to tolerate beyond this policy.

Energy Metric

As a natural approach, the primary accomplishment of the solution we are aiming at is to conserve energy. Assuming the transition to another processor performance mode at lower frequency, this means the ratio of savings in *power consumption* has to be larger than the additional proportion of *time* required for executing the same workload. Yet, it may be undesirable to run disproportionally slow for the task to complete within an equal energy budget. More sophisticated metrics like the *energy delay product* (EDP) trade an improvement in *energy* with a comparable slowdown in execution time. Energy-times-delay metrics require detailed information (derived from energy and time models) to be evaluated. Interestingly enough, research in this area discovered that such metrics, consulted on a per-interval basis, may even yield non-optimal overall results. For short timeframes, Sazeides et al. [22] demonstrate mathematically that metric values can result in suboptimal decisions compared to what the same metric would suggest regarding overall execution time.

In fact, since the amount of static power consumption during processor operation makes up a large portion of the overall power dissipation, a pure energy metric puts substantial weight on the parameter of execution time: for every second the policy affords to take longer until completion, the overall savings in energy have to compensate for the additional static energy.

Objective. *As guideline for the solution of this thesis, we define* (1) overall energy consumption *for a given workload as primary metric. In order to avoid unduly prolonged execution times, we add a supplementary constraint* (2) *not to exceed a defined* limit in slowdown.

Given that objectives (1) and (2) have to be considered at the same time, situations may arise where both goals suggest different decisions. For instance, energy savings might be possible but are prohibited by the timing constraint. As this thesis is to investigate improved energy-efficiency for the multi-core processor case, the policy definition must also hold true for multi-programmed workloads that utilize more than one processor core and may lead to conflicting situations as well.

3.4 Measurement Setup

In order to quantify the power dissipation of our target platform and make respective statements about energetic behavior, the system has been customized to accommodate the necessary measurements. By placing a serial shunt resistor into the CPU's V_{dd} rail between power supply and load (processor), it is possible for the data acquisition equipment to probe momentary power consumption (viz., the rate energy is dissipated). The actual power value can easily be approximated by the proportional relationship to the voltage drop across R_{shunt}. A very similar test setup has been used for equivalent measurements in the past [14]. It has been verified that no other components with significantly varying power characteristics were involved in the electrical circuit subject to our measurements. Further, since the sampling takes place before the power supply's output gets stabilized to the processor's final supply voltage, it is possible to derive power consumption without the need to consider changing processor voltage levels (see Section 2.1.3). This is an important fact to note as implementing our policy aims at a frequent transition between performance states, which will also entail adjustments to the processor's supply voltage.

The installed *LabView* data acquisition setup (National Instruments SC-2345) allows for voltage measurements at high sampling rates. The development environment provides access to any acquired data by means of a dataflow programming language. High-resolution data with several thousand samples per second can be used to monitor even short periods of activity, such as the execution of interrupt handling routines in the operating system kernel. However, in order to smoothen out variances in short-term activity, we created a *virtual instrument* that processes any high-rate sample input and turns it into averaged values regarding a user-defined interval. Provided that our measurements are not subject to aliasing effects, the yielded average values (calculated from a set of samples) remain sound for any energy-related considerations.

In this setup, the device under test appears as a *black box* to the researcher. Hence, any specific data or clue about the inside structure of the device regarding its energetic behavior can only be inferred by examining its crucial parameters on the outside. Obviously, the current, which might or might not flow through the separate cores of a CMP, can only be observed as a *sum* by our measurements. To develop an adequate energy model, we have to unveil the processor's basic rules in power behavior. By conducting experiments with different workloads we will approach the target systematically and then derive the general case from the observed behavior.

3.5 Basic Energy Model

The range of related work we evaluated earlier provided valuable hints about the type of workload that allows for savings in energy. According to Section 2.2.3, it is possible to distinguish between classes of instructions that are either directly affected by frequency changes (i.e., they exhibit a constant cycles per instruction behavior) or appear less affected as they depend on components external to the processor. Thus, performance of memory-intensive applications is not as much affected by frequency as applications that perform mostly on-chip activity.

Subsequently, as next step, we will analyze how the distinct functional components contribute to overall power consumption of our particular target platform. Further, we will determine how the power consumption and throughput of these resources are affected by different performance states. Our test machine exhibits two ACPI P-states to accommodate high performance (P_0 at nominal processor speed) as well as low power requirements (P_1 at reduced frequency and voltage).

In order to observe the processor's energy behavior under stress, we developed a set of simple microbenchmark applications that reiterate a particular type of operation continuously, namely utilizing *arithmetic logical units* (ALU), *speculative mechanisms*, *cache*, and *memory* extensively. Microbenchmarks invoke special cases and therefore can only provide hints regarding the behavior of real-world applications. Nevertheless, in terms of energy characteristics, these benchmarks do not exhibit the varying behavior of typical workloads and therefore can help deducing to the normal case. Further, with this synthetic behavior it is possible for particular system bottlenecks to be identified.

The `int` application performs different arithmetic operations on integer values, as does `float` on floating point values. Since the memory reference patterns generated by `l2`

benchmark	Power [W]			Performance	Energy
	$P(f_{max})$	$P(f_{min})$	ratio	ratio	possible gain
float	43.7	30.9	0.707	0.666	-6.2%
int	44.2	31.8	0.719	0.665	-8.1%
l2	45.2	32.2	0.712	0.667	-6.7%
mem	60.5	42.5	0.702	0.91	22.9%

Table 3.1: Power and performance ratios (single instance)

exhibit a high degree of locality, this application generates a large number of cache hits, whereas `mem` saturates the memory bus by copying large chunks of memory using the `memcpy()` function. We customized all benchmark application to respond to operating system signals. On a signal event, the application indicates the momentary iteration count. By sending an `XCPU` signal to the application at fixed time intervals, we can determine the current performance with regard to throughput. The large number of samples taken eliminates variations.

For the following analysis of the multi-core processor, we first study the single instance case – that is, running one benchmark task at a time on one core while the other cores are idling. We also assume that the impact of operating system management processes on all subsequent power and performance measurements is negligible. To account for the influence of temperature-related leakage effects on power consumption, we constantly ran the following microbenchmarks for several minutes prior to sampling an average power value.

By measuring power and throughput of the microbenchmarks for both available ACPI states P_0 and P_1 (at $f_{max} = 2400\,\text{MHz}$ and $f_{min} = 1600\,\text{MHz}$), it is possible to assign each application to one of the distinct *on-* and *off-chip* workload classes. As Table 3.1 shows, all benchmarks – with the exception of the memory-intensive workload – belong to the on-chip class with a lower bound of $0.\bar{6}$ in performance slowdown. This factor corresponds to the frequency ratio of $f_{min}/f_{max} = 0.\bar{6}$. At the same time, the accompanied savings in power dissipation are in the range between 0.702 and 0.719. Apart from the runtime being increased by 50%, tasks of this type of workload would not be more energy efficient until the corresponding power ratio would fall below a value of $2/3$.

As already noted, power consumption of the `mem` application follows the same rules as the rest of the benchmark programs. However, the slowdown due to lower processor clock frequency turns out to be significantly smaller. By multiplying the ratio of power savings with the factor of increased runtime (due to degraded performance), we can determine the energy savings theoretically possible to up to $22.9\% = 1 - 0.702 \cdot 0.91^{-1}$ for this synthetic case.

Based on these measurements, it is valid to assign the second level cache benchmark to the on-chip class of applications as well. This rules out cache hits as energy saver and suggests that memory references may serve as an indicator for more energy efficient operation. These observations match the results of previous work regarding different platforms and supports the notion of workload classes. It also verifies that our target platform behaves according to this classification model.

3.6 The Multi-Core Case

The following section will complement our knowledge by including the remaining cores of the quad-core processor in the analysis. In Table 3.2 and 3.3, we list the resulting data of running the same microbenchmarks on multiple cores at the same time.

int instances	1	2	3	4
$P(f_{max})$ [W]	45.0	53.0	60.0	66.5
$P(f_{min})$ [W]	31.1	35.0	39.3	43.2
ratio	0.691	0.660	0.655	0.650

Table 3.2: Multiple instances of integer tasks

mem instances	1	2	3	4
$P(f_{max})$ [W]	61.0	63.0	67.0	69.2
$P(f_{min})$ [W]	43.0	44.1	44.8	46.1
ratio	0.705	0.700	0.669	0.666

Table 3.3: Multiple instances of memory-intensive tasks

For the case of two cores fully saturated with integer tasks, the power ratio falls below the slowdown factor of $0.\bar{6}$. This indicates that it is possible to run the entire chip multi-processor at P_1 while not dissipating more energy for the same task. Further, overall decreasing power consumption ratios, irrespective of the workload, suggest that, beginning with a certain ratio of utilization, it may become profitable to scale all processor cores to a lower frequency. However, apart from violating our second policy goal to have energy savings at reasonable maximum slowdown, we would only yield marginal energy savings for `int` tasks (less than 2%). Thus, running a processor loaded with compute-bound tasks at the highest P-state only makes sense if the intention is to limit overall power consumption.

The activation of additional `int` tasks makes a distinct regularity in increasing power consumption apparent. Each instance dissipates approximately 7 Watt. This constant increase across all four configurations suggests the definition of a notional power consumption value for an *absence* of activity. Subtracting this delta from the single instance power value yields 45 Watt − 7 Watt = 38 Watt. This *baseline power* differs significantly from the actual idle power spent by a system with no utilization of approximately 30 Watt. The difference shows that the processor deactivates some common circuitry while entering a halt (C_1) state. As a result, once active, a multi-core processor should be sufficiently utilized to mitigate the burden of this activation cost. This analysis matches with the conclusion Winkelmeyer made in his work [26, Sect. 7.6].

core				Power [W]		Average	Potential
0	1	2	3	f_{max}	f_{min}	slowdown	savings
mem	—	—	—	61.0	43.0	9.9%	22.8%
mem	int	—	—	62.4	43.4	27.4%	11.4%
mem	int	int	—	68.0	44.5	34.5%	16.6%
mem	int	int	int	70.6	45.5	38.5%	17.9%

Table 3.4: Performance and power ratios (mixed instances)

Extending our analysis to configurations of mixed workloads (see Table 3.4), it becomes apparent that the activation of the shared memory bus accounts for a large amount of the overall energy dissipation. When comparing the first and the second configuration, additional `int` instances incur only a minor increase in power consumption. The power profile incurred by compute-intensive workloads is superimposed by the power required for front side bus activation and accesses to memory. The irregular rise between the second and the third configuration can be accounted to another part of the processor being activated from a low-power state. It is obvious that this processor is made from two separate parts. In fact, the quad-core design subject to our analyses comprises of two dual-core dies combined as a multi-chip module (see Section 2.1.1).

The next step is to regard the same situation when scaling the entire processor to a low frequency setting. We already know from Table 3.1 that memory-bound tasks experience a smaller performance slowdown of 0.91; in contrast to compute-intensive workloads with a slowdown of $0.\bar{6}$. Since this configuration is free of any resource contention (only one `mem` instance uses the available memory bandwidth exclusively), each individual performance situation remains unchanged compared to an independent run with the other cores idle. This means, each task executes at exactly the speed it would progress when running on one core at P-state P_1. Hence, with n denoting the number of active cores, we can describe overall package slowdown for a given configuration by using Equation (3.1).

$$\sigma(n) = \frac{0.91 + (n-1) \cdot 0.\bar{6}}{n} \qquad n \in \mathbb{N} \tag{3.1}$$

The resulting slowdown factor σ is normalized to a value of 1.0 for the same configuration running at P-state P_0. We list the results this performance model yields in the slowdown column of Table 3.4. Obviously, the penalty involved with scaling all processor cores to a lower frequency increases with the ratio changing from a more memory-bound to a more CPU-bound workload. The more the ratio is weighted towards a compute-bound workload the more slowdown we have to tolerate while energy gains diminish. For instance, energy gains of 22% that were possible with an instance of the `mem` application earlier (see Table 3.1) are lower in the second configuration and come at a high cost of nearly 30% performance slowdown. This poor performance is caused by running three quarters of the processor performing integer instructions at a low frequency.

Nevertheless, these measurements support the conclusion that a specific blend of workload – in this case, across the border of multiple cores – becomes increasingly eligible for frequency scaling the more it tends to be memory-bound. Throughout the next section, we will generalize the knowledge we obtained regarding workload blends consisting of on-chip and off-chip instructions. This approach will yield a formula allowing us to describe a basic timing model for workloads in general.

3.7 Mixed Workloads

In order to extend our knowledge to tasks that are neither memory limited nor entirely compute-bound, we decompose the execution of a given task into discrete phases according to our notion of workload classes. This may be accomplished by assigning a variable portion of time to each class. At a lower frequency, both classes contribute to the overall slowdown during their distinct phases. To an extent, this approach represents a simplification, since on- and off-chip instructions may be interleaved within the instruction stream in such a way that on-chip computations become bound by the latency of adjacent memory references. While the course of this action may result in an overestimation of on-chip slowdown, it is not an issue for the overall threshold approach as the energetically optimal threshold value will be determined by real-world workloads during optimization. As a result, the final threshold parameter might deviate from the theoretical values this section will determine. Yet, since we do not overestimate the off-chip but the on-chip workload class, we can still draw valid conclusions in terms of worst-case execution time behavior.

The previous Equation 3.1 already weighs slowdown factors on a core level, which allows us to bring this formula into a more general form. In order to do that, we introduce the *memory-boundedness* ratio ρ (ranging between 0 and 1) of a task τ. Regarding a given time interval, the value of 1 assumes a workload continuously accessing memory. For values smaller than 1, the remaining part of the interval is considered compute-bound. This yields the general form of (3.2). By separating workload phases, we can start making assumptions about the extent of impact a certain ratio of memory-boundedness may have on execution time. Obviously, with $\sigma_{mem} = 0.91$ and $\sigma_{cpu} = 0.\bar{6}$, this formula proves valid for the synthetic cases of our compute- and memory-bound microbenchmarks (see Section 3.5).

$$\sigma(\rho) = \rho \cdot \sigma_{mem} + (1 - \rho) \cdot \sigma_{cpu} \tag{3.2}$$

Putting the resulting values into relation with the power consumption ratios of our basic energy model yields a clue at which level a workload with the memory ratio of ρ becomes eligible to run at a low frequency while dissipating less energy. According to this distinction, we propose to assign tasks to two major classes of workloads: that is, either a task is *memory-bound* or *CPU-bound*. The maximum power dissipation ratio over a variety of benchmarks, determined in Section 3.5, has a value of approximately 0.72. This means the slowdown we should tolerate for any workload in the worst case – in order to still accomplish an energetic gain – should be less than this value. Within certain limits it is possible to infer from a slowdown value to the theoretical memory ratio ρ of this workload. In accordance with our model, we can use the inverse form of our slowdown model given in Equation 3.4. By solving the equation for the slowdown value of 0.72, we yield the corresponding memory-boundedness parameter necessary to not incur more than this much slowdown. Thus, for the entire range of our synthetical workloads and the given hardware platform, the premise of more energy efficient operation is guaranteed to be met with a theoretical minimum memory ratio of

$$\theta_0 = \sigma^{-1}(0.72) \approx 0.22 \tag{3.3}$$

$$\sigma^{-1}(\sigma) = \frac{\sigma - 0.\bar{6}}{0.24\bar{3}} = \frac{\sigma - \sigma_{cpu}}{\sigma_{mem} - \sigma_{cpu}} \tag{3.4}$$

By using this threshold law as a heuristic, we are able to classify a task's workload and reliably determine the most energy-efficient performance state to run this workload with. At the same time, the slowdown part of the model (Formula 3.2) predicts an upper limit for the increase in execution time. The worst case can be modeled with a workload of memory-ratio ρ tending towards θ_0. This is the workload with the longest CPU-bound phase that would yet be still classified memory-bound. Running this workload at P_1 would incur the mentioned maximum slowdown of $\sigma(\theta_0)$. With Heuristic 1, we present the basic algorithm to classify a task according to its workload type. As already motivated in the introduction, we do not regard interactive tasks in our policy. Hence, the unspecified class assignment for tasks of this type.

Since the threshold parameter is specific to the energetic behavior of a given platform, we have to verify our reasoning empirically by determining the optimal value θ_{opt} for our hardware configuration regarding a variety of workloads. In other words, θ_{opt} should allow for an optimal bisection of both task classes at runtime. We will conduct this optimization step subsequently during the evaluation of our design.

Heuristic 1 Energy profiler: global threshold policy

```
Require: $\rho(\tau)$ is the memory ratio of $\tau$, $\theta$ is global threshold value
Ensure: $\chi(\tau)$ is the task class of $\tau$
1: if $\tau$ is interactive then
2:     $\chi(\tau) \leftarrow unspec$
3: else
4:     if $\rho(\tau) > \theta$ then
5:         $\chi(\tau) \leftarrow mem$
6:     else
7:         $\chi(\tau) \leftarrow cpu$
8:     end if
9: end if
```

Multiple Active Cores

With the new notion of *mixed workloads* separated by workload class, we have a means to gain deeper insight into scenarios where tasks start to influence each other while running on a multi-core processor in parallel. In a chip multi-processor architecture, the shared memory bus is the most limiting component that is shared by all cores.

In order to motivate our reasoning, we assume up to n mixed workloads with the same memory-boundedness ratio ρ. For the sake of brevity, we will call this value *mem-ratio* for the rest of this thesis. We further assume that the memory phases of these n workloads do not overlap. Meaning, the workloads can obtain the full memory bandwidth at any point in time. In this best case scenario, all workloads run without mutual interference at maximum performance. This is possible since shared memory as well as individual computational resources are exclusively available. As the mem-ratio of ρ also denotes the duration of the memory phase per interval, accesses to memory will begin to overlap with $\rho > 1/n$. This is the configuration where the memory interface becomes the system bottleneck. Thus, within the range for ρ from 0 to $1/n$, the multi-core processor yields optimum throughput. However, with further increasing mem-ratios, individual performance will deteriorate to a value of $1/n$ once all tasks exhibit a mem-ratio of 1. Meaning, at this ratio, overall package performance equals the performance of one entirely memory-bound task running exclusively on the processor.

Yet, from an energetic point of view, heavily memory-bound workloads run more efficiently under frequency scaling. While performance slowdown for mixed workloads under scaling diminishes with increasing mem-ratios (see Section 3.2), the memory bottleneck heavily impedes overall performance the more the tasks increase their demand in memory bandwidth. Obviously, two factors with contrary influence on overall performance superimpose in this case. Hence, the optimum workload, yielding peak efficiency for a multi-core processor under scaling, cannot be one of the extreme types of entirely CPU- or memory-bound tasks. Rather, optimum hardware efficiency can be achieved for a workload having full throughput while experiencing the least slowdown effect due to scaling. This point coincides with the emergence of the bottleneck at a memory ratio of $\rho = 1/n$. The exact value of this mem-ratio depends on the number of tasks that are concurrently active. For one task, the most efficient operation point coincides with a completely memory-bound task, as performance always equals to 1. For a number of four active tasks, the optimal per-task mem-ratio is $\rho = 0.25$. If the eventual threshold parameter of our heuristic is below this theoretical value, we would exploit these points of optimum efficiency with our approach by scaling to the respective performance state accordingly.

3.8 Online Estimation

The proposed threshold heuristic of Section 3.7 requires specific performance event data in order to make determinations regarding task classes. To exploit the advantages of an online feedback approach, our design will acquire this data at runtime through performance events that account for on- and off-chip activity. In the latter case, the rate of memory accesses might serve as a meaningful indicator. This rate, however, depends on the maximum bandwidth the memory subsystem can provide and is therefore an entirely platform-specific constant, which is not known a priori.

It is necessary to determine this upper bound before we can calculate a correct memory-boundedness ratio ρ of a task τ. Further, we need a counterpart measure to account for the time the processor performs on-chip instructions. By normalizing the maximum possible memory access rate to the number of unhalted processor cycles, we yield the correct ratio ranging between 0 and 1 by simply determining the quotient of both values. With the bandwidth-saturating `mem` microbenchmark, we record up to 0.038 memory accesses per clock cycle. The inverse value is the weight we have to assign to each memory-related performance event in order to normalize it to the cycle count of 2.4×10^9 per second.

3.8.1 Memory Bottleneck

Following the analysis of the preceding Section 3.7, overall progress across processor cores can depend on the amount of contention for shared memory bandwidth.

Taking four instances of the `mem` microbenchmark as an example, we can verify empirically that each task will obtain a quarter of the available bandwidth (in accordance with the performance deterioration mentioned in the previous section). Without loss of

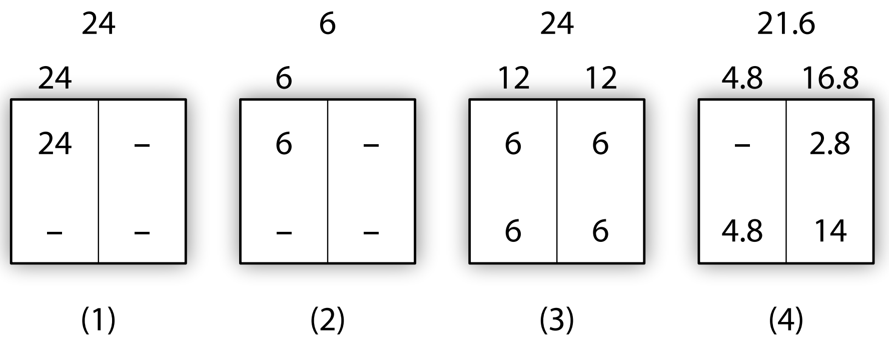


Figure 3.1: Bandwidth configurations

generality, we assume a classification threshold of $\theta = 0.3$. As also the rate of memory accesses drops to $1/4$, all tasks will be classified CPU-bound, despite of still being completely memory-bound. Obviously, the overall bandwidth situation has a significant influence on our heuristic. Moreover, CPU-bound tasks just below the threshold might become memory-bound as well, due to the increased latency for memory accesses. By turning the determined upper bound constant for memory events into a variable and decreasing it according to the overall bandwidth situation, we can compensate for the missing bandwidth on a per-core basis and eventually restore the correctness of our heuristic in general.

Consequently, we have to analyze how memory bandwidth gets assigned to distinct cores in order to apply our threshold approach for the multi-core case. We will accomplish this by developing an appropriate bandwidth model for chip multi-processors in the following section.

3.8.2 Bandwidth Model

Given the situation that more than one core performs operations on the memory bus, the controller logic has to arbitrate these accesses. Each core should be assigned bus time according to its requirements. In the most simple case we just analyzed, this means all four cores of the processor running an instance of the `mem` application will obtain a quarter of the available bandwidth.

The scenario becomes more complicated when multiple application access the memory bus at different rates. To collect clues regarding the exact arbitration behavior of a multi-core processor, we conducted further experiments with the microbenchmarks already presented throughout this chapter and a new synthetic `bench` application capable of generating mixed workloads.

We developed the `bench` application to generate workloads with alternating phases of memory- and CPU-bound characteristics. The duration of these phases is configurable by a command line parameter, allowing the user to run an application with a distinct mem-ratio. The `bench` application is intended to specifically generate synthetic workloads in accordance with the simplified notions of mixed workloads and distinct workload phases we introduced in the previous section. We will use this application to generate more differentiated scenarios for the following bandwidth analysis of the processor.

Figure 3.1 presents a set of configurations to elucidate the runtime bandwidth behavior the given quad-core processor exposes. Here, the values inside the box denote the normalized amount (see Section 3.8) of memory accesses this core issues. Each half of the processor represents a distinct core group, acting as a so-called *bus agent*. The combined number of accesses of this agent is denoted above, while the overall sum of all accesses is given by the value centered on top of the entire package.

By observing the particular behavior in a variety of configurations, we will be able to derive a heuristic, which allows to determine the remaining bandwidth for each member core. It is important to note that each core only provides access to the counters in a hierarchical fashion. All cores only have access to memory reference events counting on its own *core*, *agent* and *package* level. Thus, since the heuristic has to work on each core independently, it may only resort to this limited set of three event counts in order to infer the overall bandwidth situation. However, as an observer, we have access to all event counts at once and can therefore analyze the overall situation.

In order to gain insight and consequently work out an adequate model of the given behavior, all possible situations have to be decomposed by the influence of the individual core's bandwidth situation. For cases (1) and (2), the available memory bandwidth for each core is dictated. Saturating the memory bus in case (1) does not differ from an application that only uses a quarter of the available bandwidth: in both cases the maximum bandwidth of $bw = 24$ is applicable. The situation is different in configuration (3). This is the configuration we used as initial example. All cores run tasks that entirely saturate memory as they force the share between all cores to a 1:1:1:1 ratio. Each core obtains a quarter of the bandwidth. As already mentioned, the model must compensate by yielding a lower value of $bw = 6$ for all cores involved. Recalculating the mem-ratio using this new individual bandwidth, each core yields the correct ratio of $1 = {}^{6}\!/_{6} = {}^{this}\!/_{bw}$ (with $this$ = the decreased rate of memory accesses of a particular core). The criterion that segregates this special configuration from the former two is indicated by the memory reference count for the respective bus agent being not equal to the overall package count (see Heuristic 2, line 1).

Being the most specific case, configuration (4) allows us to infer on bandwidth assignment in more differentiated situations. We configured the `bench` application to generate a mem-ratio of $\rho = 0.25$ on two cores while running aside a bandwidth saturating application on the third core. Due to the structure of this benchmark, switching between compute and memory phases based *exclusively on timing* and irrespective of the real number of completed memory operations, two tasks with the same demand in memory accesses (ratio between CPU- and memory-bound phase) obtain a different share in available bandwidth. This fact points at the internal topology of our multi-core processor where cores of the same core group get arbitrated first. Afterwards, the requests issued by the distinct bus agents get arbitrated again. Since the memory phases of both tasks capture the same amount of time – that is, they exhibit the same amount of cycles bound to memory latency relevant to our heuristic –, the bandwidth model should compensate for the deviation in order to maintain our notion of a mem-ratio. Hence, the proposed model will assign both applications the same ratio of approximately $1/4$ by implementing different rules for both bus agents.

Since the agent on the right hand side is fully saturated, both member cores determine each others bandwidth. The core running the `mem` instance obtains a share that represents a mem-ratio of 1 for this particular configuration. The less demanding core on the same agent, in turn, obtains a quarter of this bandwidth according to its demand of $\rho = 0.25$. Thus, if we adjust the bandwidth to $bw = 14$ (i.e., the value of the higher-demanding core) for both cores, each core can again determine its correct mem-ratio (Heuristic 2, lines 5–9). As both cores cannot access the other core's memory event rate, the $core_{other}$ value has to be determined by calculating the delta between $core_{all}$ and $core_{this}$ (line 5). Consistent with the unbalanced case we just discussed, the special case of two entirely memory-bound tasks balanced matches with the case in configuration (3). The more demanding both instances become, the more the resulting bandwidths converge to a common value (e.g., $bw = 12$ or $bw = 6$ in the example of configuration (3)).

The other agent, in turn, receives a share in relation to the more memory-intensive agent. We have to approximate the rate of the most demanding core from the other side of the chip. We compensate for the differently favored bus agents by including a special case (Heuristic 2, line 11) conceding a bandwidth *bonus* (i.e., effectively a penalty on mem-ratio) of the delta between both agents. This yields the approximate bandwidth of the core with the highest demand. In relation to this increased bandwidth, the access rate of the more favored cores is in the correct order again ($4.8/16.8 \approx 0.25$). Since all these rules are experienced differently by each core involved in this scenario, the algorithm applied on all cores independently yields the correct mem-ratios for both `bench` instances and the `mem` application on the third core.

According to our analysis, we present a heuristic algorithm that covers all mentioned cases and models the processor's apparent arbitration strategy. Required input parameters for Heuristic 2 are memory-related event counts on the core, agent, and package level ($core_{this}$, $core_{all}$, $agent_{all}$) while running a specific task τ. The heuristic returns the maximum bandwidth from this snapshot in time, which can be used to calculate the correct mem-ratio $\rho(\tau)$ of the currently running task. At the same time, the algorithm is independent from the execution on a particular core. With accordance to our requirements, it properly derives the core-specific situation from the set of accessible performance counters.

Heuristic 2 Chip multi-processor: available per-core bandwidth model

```
Require: $mem_{max}$, $core_{this}$, $core_{all} \leq agent_{all}$
Ensure: $bw$ = bandwidth share (of this core), $core_{this}/bw = \rho(\tau)$
 1: if $core_{all} \approx agent_{all} \wedge core_{all} \not\approx mem_{max}$ then
 2:     $bw \leftarrow mem_{max}$
 3: else
 4:     {limited bandwidth case}
 5:     if $core_{this} < core_{other} \leftarrow core_{all} - core_{this}$ then
 6:         $bw \leftarrow core_{other}$
 7:     else
 8:         $bw \leftarrow core_{this}$
 9:     end if
10:     {agent bandwidth compensation}
11:     if $core_{all} < agent_{other} \leftarrow agent_{all} - core_{all}$ then
12:         $bw \leftarrow bw + agent_{all} - 2 \cdot core_{all}$
13:     end if
14: end if
```

3.9 Multi-Chip Module Issues

As already observed in case of the bandwidth allocation model, it is no longer appropriate to abstract from apparent dependencies and non-uniformities among cores within a chip multi-processor. Up to this point, for the theoretical considerations in terms of mixed workloads, we assumed that only the entire processor package can be scaled to a common performance state. Towards the implementation of an energy-aware policy for our platform, however, it is necessary to apply the understanding of workload classes to possible physical constraints of the particular multi-chip topology of our processor.

In Section 3.8.2, we acknowledged the existence of multiple bus agents (i.e., an agent is a group of two processor cores). In fact, this commonality extends to the capability of assuming different performance states, as in our case both cores of a dual-core die share the same clock frequency signal. Clearly, these *frequency domains* have an influence on how distinct processor cores should be scaled with accordance to the defined objectives of our policy (see Section 3.3). In order not to violate the policy definition for all member cores of a group (we will use the notions of core group and bus agent synonymously), it is necessary to coordinate workload classification of each core group. Necessarily, the performance-related decision determined for a single core earlier has to remain valid for all affected cores of a P-state change.

Assuming there is one processor core active with its workload classified as memory-bound (i.e., the task exhibits a memory reference versus bandwidth ratio greater than θ), the processor gets scaled down since there is an opportunity to save energy and the associated timing constraint is met (policy definition (1) and (2)). Without loss in generality, we extend this scenario by another core located within the same frequency domain. We assume that the new core executes a purely compute-bound workload. This in turn means that the overall ratio of compute-bound instructions for both cores surpasses the given threshold. As a result, leaving both cores scaled at a lower frequency would violate the worst case timing we assumed when scaling the first core down. Obviously, providing that it is intended to enforce both policy goals, the given threshold value has to apply to all cores within that frequency domain. Meaning, in our case, we should only scale this domain down if two tasks of the mem-bound class will get co-scheduled to run at the same time. For the remaining three possibilities, the second timing constraint would get violated. Correspondingly, the entire core group has to run at the highest frequency setting. Since processor P-states are indexed in ascending order with the highest frequency state beginning at zero (see Section 2.1.4), it is possible to translate this logical conjunction into an arithmetic **min** function over all requested performance states within a frequency domain.

Hence, in order to augment our policy to the more specific case of multi-core and multi-chip module processors, we propose the coordination of P-state requests in accordance with the objective of our policy. Consulting topology information allows to yield correct decisions for multi-chip processors in general. Algorithm 3 registers the performance state requests of all processor cores and determines the actual state of a dependent group of cores according to the **min** function. The algorithm still works correctly in the special case of a multi-core processor without any physical dependencies among cores, as in this case the set of dependent processor cores (Γ) is empty and the result of coordination coincides with the output our heuristic yields for individual cores. We amended the code with an idle policy that runs the entire package at minimum power dissipation in the absence of activity. The opt_{mem} operator returns the optimal P-state for memory-bound workloads.

Task Grouping

With the introduction of the **min** policy, it appears necessary to align execution of memory-bound tasks with the internal topology of the multi-core processor. Given a mixed configuration of `int` and `mem` tasks (see first configuration in Table 3.5), instead of running all cores at the most power consuming setting, grouping both `mem` tasks on a common dual-core die would cause the policy to scale the first core group to a lower frequency state.

Following this assumption, we measure power consumption of both configurations. Although the available memory bandwidth gets shared between both `mem` instances in

Algorithm 3 Chip multi-processor: energy and time constrained P-state coordination

Require: π^i = P-state for core i, id = processor index of this core, τ = current task on this processor, Π = set of available P-states ($\neq \emptyset$), $\chi(\tau)$ = task class of τ, Γ^i = set of processor indices sharing i's frequency domain, $\min \Pi$ = returns high performance P-state, $\max \Pi$ = returns low power P-state

Ensure: each frequency domain runs at its optimum operating point

```
1:  $\pi_{mem} \leftarrow \mathrm{opt}_{\mathrm{mem}}\,\Pi$
2:  **if** $\tau = idle$ **then**
3:      $\pi^{id}_{req} \leftarrow \max \Pi$
4:  **else**
5:      **if** $\chi(\tau) = mem$ **then**
6:          $\pi^{id}_{req} \leftarrow \pi_{mem}$
7:      **else**
8:          $\pi^{id}_{req} \leftarrow \min \Pi$
9:      **end if**
10: **end if**
11:
12: **if** $|\Gamma^{id}| > 1$ **then**
13:     {shared frequency domain constraint}
14:     $\pi^{id}_{act} \leftarrow \min_{i \in \Gamma^{id}} \{\pi^i_{req}\}$
15: **else**
16:     {optimization: no dependencies among cores}
17:     $\pi^{id}_{act} \leftarrow \pi^{id}_{req}$
18: **end if**
```

agent$_0$		agent$_1$		P-states		Power	Package	Potential
0	1	2	3	agent$_0$	agent$_1$	[W]	slowdown	savings
mem	int	mem	int	P_0	P_0	69.2	0%	0%
mem	mem	int	int	P_1	P_0	64.9	4.7%	1.8%

Table 3.5: Energetic effect of task class grouping

either configuration, we assume a full performance value of 1 for each instance, since we do not analyze this situation for a general assessment, but rather for an immediate comparison of similar configurations regarding task placement.

As each individual `mem` instance experiences a slowdown of 0.91 under frequency scaling (second configuration), we note a minor overall package slowdown of approximately 5% for the grouped configuration. Although the first agent runs at a lower frequency, power dissipation is only lowered modestly resulting in a mere energy benefit of 2% (see Table 3.5). Obviously, scaling the frequency of one half of the multi-chip module has not the same impact on energy savings once the other agent runs at P_0. We can verify this behavior by running a single task on our processor with all cores at P_1. Requesting a higher frequency setting for one of the processor cores of the idle agent results in a significant increase in power consumption, although the running task is not bound to this agent. Both bus agents apparently share the same supply voltage of a single voltage regulator. Since one of the agents operates at a high frequency, both agents are supplied with an elevated voltage level. As a result, the drop in power dissipation for a mixed P-state is not as significant (see Section 2.1.2) and the increase in energy efficiency for running two memory-bound tasks on a common agent, while the other agent operates at a low P-state, is only marginal. In fact, there are even task configurations that may result in a worsened energetic profile. Running heavily compute-bound tasks at a low frequency while the other agent is running at P_0, causes a significant performance penalty for these tasks. The minor drop in power consumption cannot compensate the overall loss in throughput. For a configuration of a `mem` and `int` task on the first agent and a single `int` task on the second agent, we determine a 16.7% loss in performance. As power consumption drops only marginally, we experience a *loss* in efficiency of up to 5.7% when running the processor at mixed P-states.

From an energetic point of view, the new constraint of a shared voltage plane does not warrant the specific migration of tasks within a multi-chip module processor for the short term. In the improbable case of an unaligned task configuration, potentially leading to efficiency loss, our **min** policy avoids energetically disadvantageous P-state configurations by its requirement of a minimum memory-boundedness of both tasks on a common agent in order to scale this agent down. The small gains, on the other hand, that might be possible with task grouping may be amortized by penalties due to increased cache miss rates after a migration.

Relaxing the Time Constraint

We developed the **min** policy with the goal to ensure both our policy objectives of increased energy efficiency in conjunction with a limited penalty on performance. By doing that, we effectively partitioned the space of possible energy savings in order to limit an exceeding increase in execution time. This means, regarding the example of Section 3.6 (Table 3.4), we intersected all configurations involving memory-bound tasks into configurations that consist only of memory-bound tasks (configuration 1, involving a minor slowdown) and configurations that represent a mix and thus cause undue runtime penalties exceeding 27%. By shifting the focus of our coordination algorithm more towards the energy objective, it seems possible to exploit more situations where energy savings are conceivable.

Considering the classification of tasks, the bandwidth model adjusts the threshold heuristic once both cores in a core group utilize the memory bus. This increases the likelihood for both cores to be classified as mem-bound. Consequently, this frequency domain can be scaled down. Yet, as soon as another compute-bound task is added, all scaling gets inhibited due to the strict timing constraint. By relaxing that constraint, we can exploit the latter configurations (Table 3.4) that were energetically beneficial but incurred significant performance penalties on the package level. In order to consider the other half of this partitioned space with our threshold policy, we can shift the coordination policy from a logical conjunction to a disjunction. With regard to our algorithm this means the implementation of an arithmetical **max** function over all performance state requests. The rest of our framework for state coordination remains identical. The choice between policy objectives leads to another configurable parameter for our design. It allows the user to specify that – under the given hardware constraints – either timely execution should be favored or opportunities of possible energy gains.

Obviously, in the single-core case, the results of both proposed algorithms match. Given that only one core is active, it is not necessary to distinguish both policy goals under the constraint of shared frequency domains. However, due to the shared voltage plane of our multi-core processor as another constraint in freedom, it only makes sense to apply a **max** policy if core groups can actually run at a low frequency and voltage. Running both agents at mixed performance states would invoke the case of decreased efficiency we determined earlier (see previous Section 3.9). Hence, for a chip multiprocessor similar to our hardware platform with voltage dependency, the energy favoring **max** policy should be applied on the entire set of dependent cores (i.e., all cores of the processor package) to yield optimal gains in energy.

3.10 Energy-Efficient Scheduling

In the preceding sections, we systematically analyzed the energetic behavior of the target processor in order to develop a strategy to run a multi-core processor continuously at an optimal performance state. We will pick up the threads of these sections to finish the proposed design and concludingly summarize the advantages and limitations this solution has.

According to Bellosa (see Section 2.2), task-specific information about energy-related characteristics is necessary in order to allow the operating system to implement sophisticated power management policies. We follow this approach by building our solution on an online energy estimation design of previous work [18]. Determining energy-related information at runtime, however, is based on the assumption that it is possible to extrapolate from past to future behavior and thus carries the risk of potential mispredictions. There may even arise pathological situations, where program behavior switches between opposite workload types with each estimation interval. Since the decision for the upcoming time quantum will be based on the classification of the previous one, this pathological behavior would result in all decisions being contrary to the actual situation. However, assuming that changes in program behavior happen less frequently (i.e., the duration of a time quantum is much smaller than the period of an application phase), this best-effort approach will yield a correct determination for a majority of the runtime.

As input for the threshold heuristic, specific data regarding workload classes is required. The heuristic is responsible to determine the class a task belongs to (CPU-bound or memory-bound). In order to yield correct results, the processor's shared memory bandwidth has to be taken into account. Prior to being evaluated by the threshold heuristic, data regarding the off-chip workload class is edited by the bandwidth heuristic. The flow of information in this design is depicted in Figure 3.2.

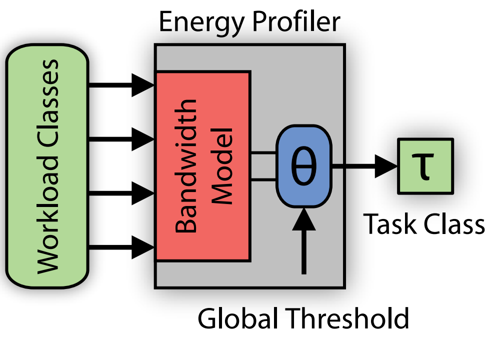


Figure 3.2: Workload classes and energy profiler

By employing a heuristic, we can avoid the need for a complex energy model. Such a model for all available P-states would be able to estimate a workload's exact energy requirements at different performance states. Thus, more differentiated decisions regarding energy gains and execution times would be possible. However, maintaining such a precise model will most likely incur considerable runtime overhead due to the calculation of energy values for a multitude of state configurations, especially in the multi-core case. We argue that the extra effort would only result in minor gains for corner cases. With our best-effort approach, on the other hand, we differentiated the significant cases during analysis and accordingly derived generally valid conclusions: we moved the calculation overhead from the latest possible moment at runtime to our offline analysis. Still, heuristics are a powerful tool to limit the complexity of decisions while promising comparable gains to more elaborate approaches.

In our case, the heuristic requires a simple input parameter, which allows the user to tune the trade-off the policy is allowed to make. Increasing the threshold parameter gives the user the opportunity to dictate more strict guidelines in terms of timing requirements. Still, on the downside of this approach is the need for determining the optimal threshold value that matches the energetic characteristics of the given hardware platform.

In order to accommodate the propagation of workload class data and the inclusion of the bandwidth model and threshold heuristic, the energy infrastructure has to be augmented in its estimator and profiler component. Since consequently energy-related information is directly associated to the task's execution context, the operating system scheduler instance running on a particular processor core can issue the request to adjust the performance state for the following time quantum.

The proposed coordination framework processes this request according to hardware topology and policy requirements. The resulting overall system design and the relation between components is depicted in Figure 3.3.

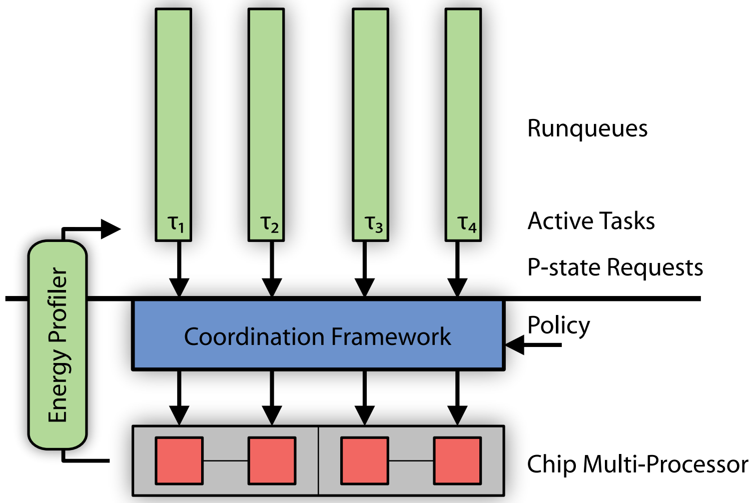


Figure 3.3: Energy-efficient scheduling and P-state coordination

The indirection achieved by this underlying coordination component represents a layer of abstraction from hardware peculiarities. The user can specify on a high level, which policy objectives have to be implemented. Irrespective of possible performance state dependencies among processor cores, coordination ensures at any time that all objectives are enforced accordingly.

Chapter 4

Implementation

In order to implement the design we propose, our solution builds on the *Energy Infrastructure* (see Section 2.2.4) for the Linux 2.6 kernel developed by Merkel [18]. The available code provides the basic building blocks to perform online energy estimation based on performance counter events. The energy profiler assigns this energy-related information to the currently running task. We extended the infrastructure in the respective components that either had to be modified to work in conjunction with our processor architecture or augmented to process the information necessary to our model.

Both, our platform independent and the architecture-related modifications for the Core 2 microarchitecture sum up to several hundred lines of code. The rest of the system remained unchanged, since we aimed at demonstrating that it is possible to achieve an optimized energetic behavior of the system by applying a small set of modifications. Apart from the architecture-specific details we will consider during the course of the this chapter, our design should be generic enough to be ported in a similar form to other operating systems. The accompanying algorithms are available in pseudo code (see Chapter 3) and thus ready to be implemented irrespective of any particular programming language or platform.

4.1 Energy Infrastructure

The code by Merkel has originally been developed for multiprocessor systems based on the NetBurst microarchitecture (Intel Pentium 4). These processors offer a large number of performance counters that can be programmed to trace events in different functional parts of the chip. The Intel Core 2, however, follows the design inherited from the Intel Pentium Pro (P6) generation of processors. This architecture provides two performance counter registers, which can be programmed independently to count two types of events at a time.

4.1.1 Architecture-Specific Enhancements

For the energy estimator as architecture-specific part of the energy infrastructure, we had to apply enhancements that accommodate the new performance counter facilities of our target platform. Although available performance counter registers are also mapped into the larger address space of model-specific registers, newer generations of microprocessors ship with an optimized instruction set to access performance counters. The `rdpmc` instruction of the Core 2 requires less processor cycles to read a performance register and allows to access the available pair of programmable counters.

The programming of the energy estimator with event counters and their respective weights is typically done from user space in order to make the kernel mechanism applicable to a range of similar performance counter architectures. By means of a formatted string passed to the `Sysfs` filesystem, the estimator can be instructed to count a specific event. We extended the estimator code with support for the `rdpmc` instruction and introduced a new syntax to make this functionality accessible. As only two independently programmable performance events are available with the given hardware architecture, we also implemented *multiplexing* for performance counters, which allows to use more energy weights than counter registers actually implemented in hardware.

Multiplexing is accomplished by a time-sharing approach, changing the programming of the event select register (associated with a performance counter) in a round-robin fashion. On each timer tick, the respective counter updates the event count for the elapsed time interval and proceeds with counting the next event in the list. Further, to maintain the context of each event counter to the task currently running, this switching has to occur on every task switch as well. However, depending on the number of events being multiplexed, it is necessary to extrapolate the event counts to a full time interval. Hence, the accuracy of counter multiplexing is limited by the rate of timer ticks and the duration of the minimum scheduling timeslice. In an extreme case, a task could toggle its behavior with every timer tick, resulting in the estimator only counting events of alternating program phases and not the overall task behavior. To allow for fine-grained performance counter multiplexing, we configured our kernel code to a timer frequency of 1000 Hz. In the case of task preemption due to the activation of higher priority tasks in the system, we discard multiplexed event counts from time periods that are too short to yield meaningful results from extrapolation and consequently would provoke bogus values.

Performance counter multiplexing compensates for the limitations of the underlying hardware architecture on a low level, without influencing the remaining part of the design. Apart from the activation of multiplexing by including the `mux` keyword in the programming string for a new energy weight, the particular method of how performance counters are evaluated by the energy estimator is transparent to the rest of the energy infrastructure.

4.1.2 Workload Classes and Threshold Policy

We implemented the bandwidth model and the threshold classificator in the energy profiler according to the pseudo code of Chapter 3. The energy infrastructure extends the operating system's task context with a data structure representing the energy profile. In order to support our algorithms, building on the notion of workload classes, we extended this energy profile with a generic vector of energy values. Workload classes are a generalization of the concept of a single value representing the task's estimated energy consumption. We now have a set of weighted energy values specific to classes of energy-related events. The underlying energy estimator has been enhanced to assign arbitrary sets of energy weights to each class. By introducing a vector of bits for each energy weight, we can specify to which workload classes this weight contributes to (e.g., a bitmask of `0x6` means this energy weight contributes to the second and the third workload class).

This results in a more generic concept of a linear combination of energy values split into multiple classes. These energy classes, in turn, provide the necessary input data for the subsequent bandwidth and threshold heuristics according to our notion of compute-related and memory-related events. With the freedom of configurable energy weights and class assignments, our mechanism can be flexibly reconfigured to match the energy-related properties of a variety of architectures.

In order to support our threshold classification heuristic, we require two separate class inputs accounting for the time the processor spends with the distinct workload classes we defined as rationale for our solution. We use the number of unhalted processor cycles as a measure for the time spent with on-chip activity. The performance event of `UNHALTED_CYCLES.REF` counts active processor cycles according to a reference clock signal, which is unaffected by frequency scaling, thus providing a constant measure per time interval for our on-chip class. Regarding memory accesses, we require a type of event, which can be counted on the hierarchical levels of a chip multi-processor topology. We select the performance event `BUS_TRANS_MEM`, which can be programmed to count memory accesses on the core, agent, and package level. In order to support the bandwidth model and subsequently the correct classification of the task workload, we have to propagate all four event rates as abstract workload classes in the task's respective energy profile. The global threshold parameter is incorporated into the energy profiler and can be configured via the `Sysfs` file system interface. Reevaluating the workload classes on each context switch enables the energy infrastructure to determine the current task class for each process according to its recent behavior.

In order to simplify the process of configuring the energy estimator according to our policy, we developed a user-level program `core_pmc` that can generate the appropriate bitmasks for programming performance monitoring counters. Complemented by a set of shell scripts, we configure the necessary workload classes required as input for the bandwidth and threshold heuristic.

4.2 Coordination Framework

With the introduction of our *coordination framework* for processor ACPI states, we bypass Linux' CPUFreq software infrastructure usually responsible for CPU power management. CPUFreq is geared towards implementing basic policies and is not properly prepared to be called from core parts of the operating system as the scheduler. Since the scheduler is a frequently invoked part of the operating system, changing the processor's performance state should incur the lowest overhead possible. However, compared to the CPUFreq code, it is not necessary to migrate the current thread of execution to the processor that is intended to change its performance state. As each scheduler instance runs exclusively on the processor it has been assigned to, moving the responsibility of issuing P-state requests to the scheduler makes sense, as now the necessary core-specific instruction can be invoked directly.

In order to realize a valid implementation of the proposed coordination algorithm (see Chapter 3), we first have to determine the exact behavior of our multi-core processor when different performance state values are written to the respective core's P-state MSR. In contradiction to the platform datasheet, stating that the processor assumes the lowest (i.e., highest frequency and highest voltage) of all requested P-states, each agent of the package transitions to the last state requested. That is, issuing a `wrmsr` instruction on either of both member cores transitions the entire agent to the new state. Nevertheless, the documented behavior applies to the shared voltage plane between agents, where the package transitions to the highest voltage required to operate both agents according to their frequency requirements. Consequently, the proposed **min** and **max** policies can be implemented by noting all current requests, calculating the respective minimum or maximum value, and selectively issuing the necessary P-state transition if the new state differs from the actual state.

By using a *domain mask* for the calculation of minimum or maximum P-state, we can realize coordinated behavior on an arbitrary set of member cores. According to our reasoning of Chapter 3, the domain mask should reflect the physical dependencies of the chip multi-processor. Given the particular behavior of our target platform, a core level coordination would result in erratic behavior as all state requests issued on a frequency-dependent core group would cause this agent to toggle between states. Meaning, for both policies (**min** and **max**) all coordination should at least happen on the core group level. This core group information (input parameter Γ of Algorithm 3) is derived from the operating system's knowledge about physical core dependencies. The set is empty once there are no P-state dependencies between cores (native multi-core processor). In this case, coordination is bypassed and performance state requests are issued without need for reevaluation. Further, as stated in Section 3.9, using a **max** policy for a chip multi-processor with shared voltage plane only makes sense if the entire package can scale to a low voltage. Running an agent at low frequency but high voltage is suboptimal in terms of energy efficiency.

In order to hide these platform peculiarities, we provide a high-level interface function to our energy-efficient scheduling policy, which allows to set the main objective to either favor *time* or *energy*. According to this high-level objective, the coordination framework is configured to use the arithmetic **min** or **max** function (see Chapter 3). Following the rules mentioned above, the domain mask defaults to a value for core group coordination, unless the chip multi-processor features a shared voltage plane. In this case, if energy is the primary policy objective, the domain mask is set to all member cores of the package.

4.3 Scheduler

We modified the Linux scheduler in a way that it calls an energy-related function on every context switch in the `schedule()` function, dispatching a performance state request to the underlying coordination framework. Representing a layer of abstraction from processor hardware, the coordination framework provides a generic interface for the scheduler to the underlying core's performance states. Once the general energy-aware policy objective is set, the scheduler can pass performance state requests to this layer according to the requirements anticipated for the task regarding the next timeslice. The scheduler itself remains oblivious from any possible physical hardware constraint regarding performance states.

Our scheduler activates the `sched_mc_power_savings` policy of the Linux 2.6 kernel for a basic load balancing strategy. In compliance with our previous observations in Section 3.6, this policy implements energy-aware load balancing striving to first utilize entire core groups before assigning tasks to another core group or package. While this behavior can lead to suboptimal usage of available hardware caches [23], such a strategy saves possible activation costs in energy for a second core group. Furthermore, both our coordination strategies are favored, since configurations of two tasks will be co-scheduled on one core group, deferring situations of energy inefficiency due to a shared voltage plane to configurations where both agents necessarily have to be active.

Chapter 5

Evaluation

For this thesis, performing the evaluation of the proposed algorithm serves two purposes. Primarily, we want to demonstrate that the approach we suggest can in fact significantly benefit overall energy consumption. These gains were indicated earlier by our preliminary observations with synthetic workloads. However, these theoretical scenarios are improbable to happen to this extent in real-world applications. Thus, the results we may expect are likely to be more moderate than the theoretical gains of a best case. Secondly, the measurements will serve as necessary optimization step for our heuristic where we determine the optimum setting of the main classification parameter. An important property of this parameter should be the correct separation of compute-bound from memory-bound workloads.

5.1 Test Setup

The configuration of our test machine is based on a commodity multi-core hardware system. An Intel Core 2 Quad (Q6600, Kentsfield) with 4 MiB of L2 cache and 4 GiB of RAM was running on a desktop mainboard. In order to evaluate and optimize our policy, we conducted energy measurements running the SPEC CPU2006 suite of benchmarks. The measurement equipment was the same as presented in Chapter 3.4.

To account for the overhead our modifications may incur, we used a vanilla Linux 2.6.18 operating system kernel for the collection of reference values in execution time and energy. In order to compensate for minor variations in benchmark runs, we collected the data of five runs and averaged the results. All remaining measurements have been conducted with our modified `-ees` branch of the Linux kernel. Accordingly, all subsequent gains in energy efficiency will be calculated against a possible increase in execution time due to the overhead of our policy.

Prior to benchmarking, however, we will prove by measurement that the bandwidth model of Section 3.8.2 yields viable results for our target system. Based on that model, we proceed determining the energy gains possible for the single instance case across a range of real-world benchmarks. Subsequently, we will evaluate whether our design performs as expected for the utilized case (i.e., a mixed set of tasks running at the same time).

5.2 Bandwidth Model Verification

In order to verify the soundness of the proposed bandwidth model, we ran multiple configurations of synthetic benchmark tasks comparable to the examples we resorted to during the modeling process. By means of the `procfs` pseudo filesystem interface, we have access to the information held within each task's energy profile. This includes real-time information about workload classes and the task class this process has been assigned to by the classification code.

In accordance with our design, the memory-saturating `mem` microbenchmark is reliably classified as memory-bound. While the application is running, the determined mem-ratio never drops below a value of 0.95. Compute-intensive tasks, on the other hand, show a mem-ratio of 0 as no references to memory occur except for the phase of task creation. This confirms basic operation of the proposed estimation and threshold design including the required enhancements.

In order to check the bandwidth heuristic in conjunction with the classificator, we have to produce cases of serious utilization imbalances between bus agents. Regarding configurations of concurrently running tasks, we tried to determine the threshold value that is necessary to ensure that the entire set of tasks still receive correct classification. As the real value is known, it is possible to infer on the accuracy of our underlying bandwidth model in a real application.

In case of four instances of `mem`, a threshold value of 0.95 makes sure that all tasks are classified memory-bound. In order to include the correct classification for the case of maximum asymmetry in our model, the threshold value had to be lowered to 0.85 to ensure the same premise for three running tasks. While the energy infrastructure is subject to systematic fluctuations, this translates into a margin of error of up to 15% for our heuristical approach. For more balanced situations, however, the deviation is less than 5% (see above), verifying the soundness of the heuristic bandwidth model we built for this processor architecture.

5.3 Benchmark Suite

Apart from the microbenchmark applications we used to develop a basic energy model, we will evaluate our design with the SPEC CPU2006 suite of benchmarks in the following sections. These benchmarks represent a set of standardized workloads, which have been derived from real-world applications and also exhibit a more variant behavior compared to the static properties of synthetic microbenchmarks. This is important as we want to prove that the proposed design based on online estimation reacts accordingly to changing application behavior.

As a first step, we present the particular set of benchmarks we used for our evaluation. In order to assess the nature of these benchmarks, we use the inverse representation of

Benchmark	execution time [s] f_{max}	f_{min}	mem-ratio hint $\rho(\tau)$	Description
400.perlbench	605	874	10.5%	Programming Language
401.bzip2	858	1250	8.1%	Compression
403.gcc	563	759	30.9%	C Compiler
445.gobmk	800	1163	8.7%	Artificial intelligence
456.hmmer	1225	1819	2.8%	Search gene sequence
458.sjeng	923	1342	8.7%	Artificial intelligence
462.libquantum	1357	1748	45.1%	Quantum computing
464.h264ref	1224	1815	3.2%	Video compression
471.omnetpp	591	717	64.8%	Discrete event simulation
473.astar	778	1081	21.8%	Path-finding algorithms

Table 5.1: Set of SPEC CPU2006 benchmarks (reference values)

our timing model (Formula 3.4). In accordance with our prior usage of this equation, we can determine a hint regarding the ratio of memory phases for each task (i.e., the amount of memory-related slowdown). The result will also suggest which performance state would be most beneficial for the benchmark to run with.

A large portion of the benchmarks in Table 5.1 show a time penalty close to the theoretical value of 1.5 for on-chip workloads. Clearly, these benchmarks are almost entirely CPU-bound and should therefore run at the fastest performance state. There are also the instances of `libquantum`, `omnetpp`, `gcc` and `astar` that seem to be rather memory-bound as the penalty factor is more moderate and tends towards the value for off-chip activity. Regarding memory ratio hints, we have to note that these values relate to the entire execution time of the benchmark. For example, a given ratio of $\rho(\tau) = 21.8\%$ can only model the *average* mem-ratio of this task, meaning that we cannot make a determination whether the real application behavior consists of short memory-intensive phases or a constant rate of memory accesses.

5.4 Parameter Optimization

In this section, we attempt to determine an optimal threshold parameter for the combination of our software design and the particular hardware system we run the benchmark applications on. The operating system kernel is configured with the workload class weights normalized appropriately (see Section 4.1.2). We will measure timing and energy dissipation with the installed equipment while the presented set of SPEC benchmark applications run on the test machine. After each iteration, the global threshold value will be adjusted in order to determine the influence of this parameter on overall energy efficiency. As we compare measured energies and execution times with the results of an unmodified kernel at the highest processor frequency setting, energy gains can assume a negative value for additional overhead and mispredictions, approximately zero for CPU-bound tasks that are correctly classified, and a positive value for tasks that are running at a more energy efficient performance state due to our improved scheduling policy. For the case that the threshold parameter is tuned optimally, the average value of energy gains across all benchmarks assumes a maximum value. This average value is of no particular expressiveness except from serving as an indicator for the optimum setting and, further, being consistent across the series of measurements we conducted for this optimization process. The following diagram (Figure 5.1) depicts these average energy gains varying with a threshold parameter ranging between 0.1 and 0.7.

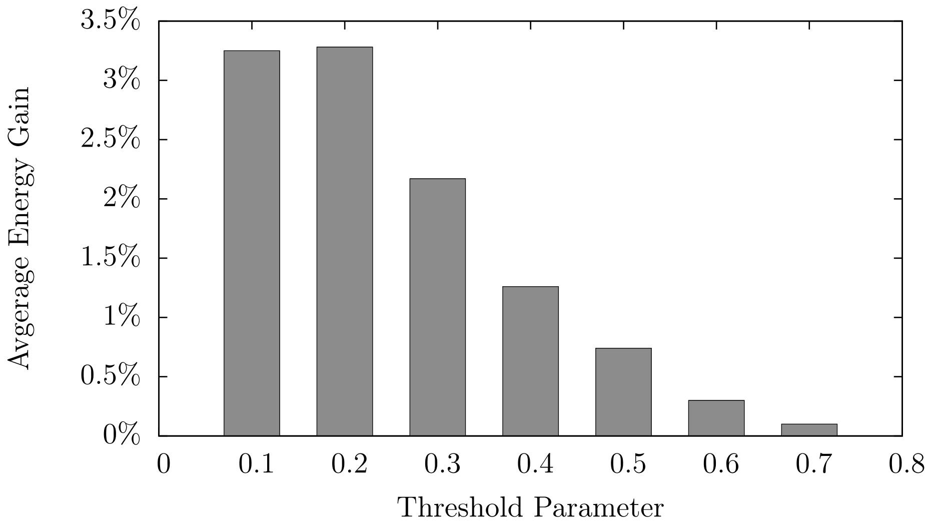


Figure 5.1: Average energy savings depending on threshold parameter

The diagram shows that average gains do not vary significantly within the range of 0.1 to 0.2 for the threshold parameter θ. Detailed results with the lower threshold value ($\theta = 0.1$) show that we have to accept a slightly increased runtime for applications that have a mem-ratio in the range of the threshold parameter. Although average energy gains are similar, possible mispredictions that have a beneficial effect on energy gains in some cases incur performance penalties in others. As the accuracy of separating workloads is slightly better for the higher threshold value and this choice also shows the most reliable energy gains across the set of benchmark applications we evaluated, we select an optimal threshold parameter of $\theta_{opt} = 0.2$. Although covering more extreme cases induced by synthetic microbenchmark code, the theoretical value of $\theta_0 = 0.22$ we determined in Section 3.7 is in the same order as our θ_{opt}. Obviously, the real-world workloads of the

SPEC benchmarks exhibit a higher degree of memory dependence among instructions, making it profitable to scale at a lower threshold value in practice.

In Table 5.2, we directly compare the reference values we determined earlier with the results in energy and execution time for a threshold of θ_{opt}. According to the gains listed in this table, it is possible to save up to 13.8% in energy for a single memory-intensive application. Consistent with the assumption of Section 5.3, the `omnetpp` benchmark heavily relies on the memory subsystem and therefore yields the best results in energy gains. Second to this case is the `libquantum` benchmark with 9.35% gain in energy. These memory-intensive applications experience a performance slowdown of up to 29%, whereas the CPU-bound benchmarks consistently run at full speed. Although the applications with energy savings run substantially longer, this maximum slowdown is well below the theoretical worst-case prediction of 40% slowdown our timing model (see Section 3.7) yields for $\theta = 0.2$.

With a slowdown of 29%, the `libquantum` benchmark would violate the timing constraint for a threshold value of 0.5. Our measurement for $\theta = 0.5$ shows that the increase in execution time for this benchmark drops significantly to 18%. Obviously, the timing constraint is still held, resulting in a slowdown that remains within the allowed range for the new threshold value. However, as opportunities to adjust the processor performance state are severely limited by a higher threshold, energy gains drop to 6% accordingly. In accordance with more strict timing requirements, the higher θ is chosen the more overall energy gains diminish (see Figure 5.1).

Having determined the optimum threshold parameter that separates workload classes reliably for a variety of benchmarks, we will proceed to evaluate the case where multiple tasks run concurrently on different cores of the processor. We will keep the optimal threshold setting of $\theta_{opt} = 0.2$ for all remaining measurements. For the single-core case, it has to be noted that the policy works in accordance with our model by enforcing the timing constraint and running both task classes at their most efficient performance state.

5.5 Multi-Core Energy Gains

As next step, we proceed in verifying the correct implementation of our policy for the multi-core case. According to the exact policy goals we specify at runtime – energy efficiency regarding or disregarding a time limit – the underlying coordination framework gets adjusted appropriately. In our case, with a multitude of hardware dependencies inside the multi-core processor, state coordination is a necessary requirement to ensure that the processor runs efficiently within the bounds of the given policy. As already noted during design in Section 3.9, both coordination policies match for the single-core case. This is the reason why we were able to disregard the coordination policy in the previous section.

For our evaluation, we present some exemplary configurations, demonstrating how the

	Reference Runs (f_{max})		$\theta_{opt} = 0.2$	
Benchmark	energy [J]	time [s]	energy gain	time factor
400.perlbench	30238.7	605	0.55%	0.99
401.bzip2	43738.7	858	1.32%	0.99
403.gcc	29203.6	563	3.02%	1.04
445.gobmk	39150.9	800	0.09%	1.00
456.hmmer	60474.7	1225	0.19%	1.00
458.sjeng	45978.9	932	-0.05%	1.00
462.libquantum	82383.9	1357	9.35%	1.29
464.h264ref	58840.4	1224	-0.12%	1.00
471.omnetpp	32615.2	591	13.81%	1.22
473.astar	40320.9	778	4.68%	1.04

Table 5.2: Energy gains and time factors for θ_{opt} (SPEC 2006)

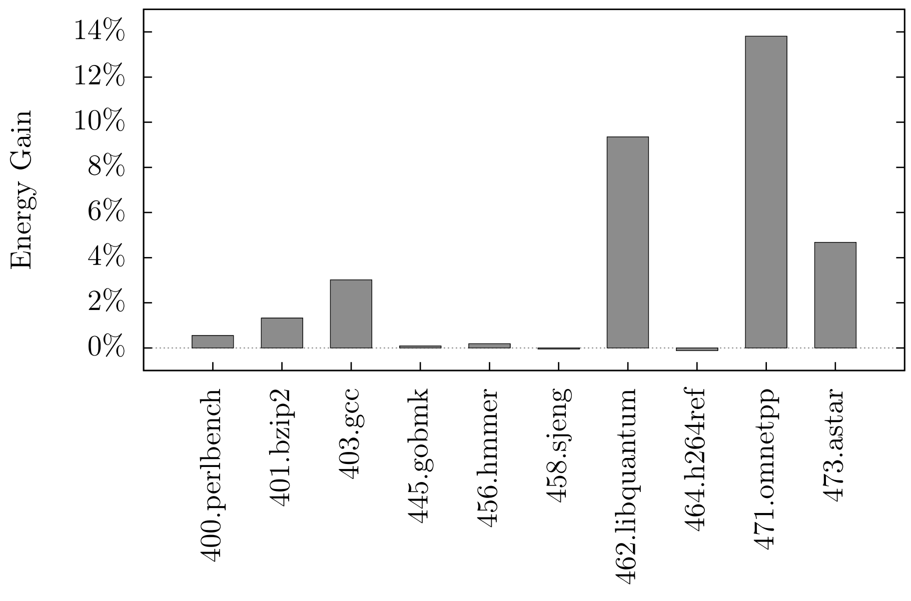


Figure 5.2: Single-core energy gains for θ_{opt} (SPEC 2006)

core				Policy: time		Policy: energy	
0	1	2	3	energy gain	runtime	energy gain	runtime
libq	hmmer	—	—	1.34%	1.04	11.9%	1.26
hmmer	libq	hmmer	—	0.29%	1.0	14.41%	1.23
gcc	gcc	gcc	—	1.34%	1.03	6.73%	1.10
libq	libq	libq	—	6.81%	1.03	7.34%	1.03
libq	libq	libq	libq	26.58%	1.06	27.0%	1.06

Table 5.3: Multi-core energy gains (regarding time and energy policy)

given policy gets enforced with regard to the overall processor package and proving that savings are also possible in the utilized case. We chose a subset of the available benchmarks that belong to different task classes but have comparable execution times. The `hmmer` benchmark is a CPU-bound application, whereas `libquantum` is heavily and `gcc` moderately memory-bound. For a multi-programmed benchmark, we start all processes at the same time. The measurement will proceed until the last process finishes. With an output of overall elapsed time for the entire job, we can determine the interval of measurement samples that have to be considered for this run.

In Table 5.3, we compare test results of various combinations of tasks with the multi-core coordination policy either set to favor *time* or *energy*. The column for the time policy shows that notable energy gains are possible if the task configuration allows to scale the processor without the risk to overly increase overall runtime. This is possible if all tasks are memory-bound, resulting in a lower demand in energy of 6.8% for three instances of `libquantum`. Compute-bound configurations, on the other hand, are executed with regard to optimal runtime behavior and thus allow for no improvements in energy dissipation. Accordingly, this restrictive policy limits the maximum runtime penalty to 6% across all configurations. For consistently memory-bound configurations, where such a strict timing can still be realized under scaling, the policy yields the respective energy efficiency.

Relaxing the requirements on timing, it is possible to save energy in more situations. For configurations where a CPU-bound task formerly inhibited the scaling of the chip multi-processor due to the timing policy, we can now note considerable gains in energy. For three tasks, consisting of `hmmer` and `libquantum` instances, gains reach a level of 14.4%. However, this efficiency improvement is accompanied by a runtime penalty of up to 26%. For configurations of consistently memory-bound tasks, the results for both policy objectives match.

Notably for the moderately memory-bound `gcc` benchmark is that a configuration of three instances executed in parallel can yield more significant gains of up to 6.7% than compared to the minor gain of 3% in the single-core run (see Table 5.2). In the extreme scenario of four memory-intensive tasks contending for shared memory bandwidth, our concept of energy-aware scheduling can exploit the full efficiency of the multi-core hardware platform subject to our analysis. At a tolerable increase of 6% in runtime, it is possible to decrease energy consumption by 27% compared to an operating system not using ACPI P-states.

If the user is willing to trade optimum performance for increased energy efficiency in the multi-core case as well, it seems appropriate to recommend a default configuration of our energy-aware scheduler with $\theta_{opt} = 0.2$ and a coordination policy favoring energy in order to yield energy-optimal system behavior.

In summary, both coordination policies enforce their respective high-level objectives. The energy policy increases efficiency notably for a variety of workloads – however, in most cases with a significant impact on performance. Unless overall workload is not heavily memory-bound, the time policy is enforced strictly, resulting in a behavior where scaling of the multi-core processor is only allowed if the incurred delay can be guaranteed to be very limited. This, in turn, is also dependent on the value of the global threshold parameter. Furthermore, the distinction between policy objectives is a result of the imposed hardware constraints in the first place. An underlying hardware platform free from performance state dependencies would allow for more fine-grained decisions as frequency domains would be more fine-grained as well. Still working correctly, we assume that our energy-aware scheduling policy would lead to more differentiated results on such a platform, with energy gains and time penalties ranging within the bounds of the two coordination policies we evaluated. Conversely, assuming a processor architecture with further accumulated frequency domains of four or more processor cores and only two P-states, the results of both policy objectives would diverge even more due to the inherent limitations of the underlying hardware with regard to effective use of frequency scaling.

Chapter 6

Conclusion

6.1 Summary

Recent generations of microprocessors ship with the capability to transition between a set of performance states at runtime. The dynamic adjustment of clock frequency and processor supply voltage has a significant influence on power consumption as well as performance. However, the impact on performance is not as severe for activity that is bound to the latency of components external to the processor (i.e., main memory or off-chip bus systems in general). Given that the processor continuously runs at its highest frequency and voltage setting, the accompanying waste of opportunities to save energy applies to traditional single-core and multi-core architectures alike. Although methods to exploit more efficient processor operation have been developed in the past, viable solutions that realize such an approach for recently deployed systems, prevalently based on chip multi-processors, are not available for use with commodity operating systems. Clearly, as the trend towards more sophisticated processor architectures will continue, power management has to keep up with the complexity of an increasing number of cores within a single processor package.

In order to tackle the challenge of designing an energy-aware power management policy for multi-core processors, we based our analysis on notions that have been introduced by previous approaches to CPU power management. The distinction between two main classes of instructions, on-chip and off-chip – categorized by the different performance under clock scaling –, leads to memory accesses as an indicator for inefficient use of energy at a high clock frequency state. As foundation for our solution, we exploit a heuristic in conjunction with an online estimation design, which collects energy-related information by means of hardware performance counters. The heuristic provides the mechanism to reliably determine the energy-optimal performance state regarding a single processor core. At the same time, the mechanism remains independent from any particular hardware, as the threshold parameter to this heuristic is configurable according to the specific energy-related characteristics of the given processor architecture. The optimal parameter to this heuristic has to be obtained by measurement.

For architectural reasons, chip multi-processors feature a series of shared resources among its constituents. These shared components, most prominently the memory bus and common frequency/voltage circuitry, prove to pose the main obstacles for the implementation of an energy-aware scheduling policy. Since available memory bandwidth is shared between cores, the rates of memory-related performance events heavily depends on the interaction between member cores. We oppose this problem by employing a bandwidth model that compensates for the mutual interference and secures the correct performance of our workload classification. Further, as the transition to a different frequency and voltage state may influence adjacent processor cores, our system coordinates performance state requests in order to ensure that workload execution for the entire clock or voltage domain does not violate or even invert the given policy objectives.

Based on an existing energy infrastructure implementation for the Linux 2.6.18 kernel, we extended the energy estimator component to support multiplexing of the limited set of available hardware performance counters. Multiplexing remains transparent to the rest of the infrastructure while the intrusive support of workload classes is necessary to support the bandwidth and threshold heuristics located in the energy profiler. By evaluating our implementation with a set of benchmarks, we were able to tune the hardware-specific threshold parameter of our policy for optimal gains in efficiency while running on the Intel Core 2 Quad (Q6600) based test machine. Running compute-intensive tasks at a low frequency results in an undue increase in execution time and typically a worsened energy profile. Accordingly, our policy schedules CPU-bound workloads with optimum efficiency at a high frequency setting. Compared to a system constantly running at nominal processor frequency, our implementation increases overall efficiency for memory-intensive applications and in idle phases. Providing that the typical workload situation for a multi-core system consists of a diverse range of applications, as in case of the benchmark suite used for evaluation, improvements of 14% in energy efficiency for single and multiple task workloads are possible. Given all tasks are heavily memory-bound while the processor is fully utilized, our energy-aware scheduling design can harness the full efficiency of the processor hardware leading to an 27% decrease in energy consumption. While these gains may come at a considerable penalty of execution time increased by a quarter, our policy can be configured to favor time in situations where performance is indispensable. With this objective and a mix of CPU- and memory-bound applications, energy savings of up to 7% are possible.

6.2 Achievements

In this thesis, we demonstrated the potential of multi-core processors to handle given workloads more energy-efficiently. Compared to the very basic power management policies shipped with today's operating systems, our approach considers specific information about workload composition and processor topology at runtime to determine the most efficient hardware performance state configuration this workload should run with. There are two configurable parameters to the presented policy allowing the user to influence the trade-off between increased runtime and energy gains.

We built our solution on an existing energy infrastructure and enhanced this design in order to accommodate a new processor architecture and the algorithms we developed. Exploiting the concept of workload classes, we provided two major contributions towards an energy-efficient scheduling policy for multi-core processors. Firstly, a bandwidth model ensuring that the proposed threshold heuristic yields correct results in situations where a shared memory bus becomes a bottleneck. Secondly, a coordination framework that accounts for the clustered architecture of chip multi-processors and maintains the given policy objective for the entire CPU package.

By modeling the behavior of chip multi-processors and mapping this behavior to heuristic algorithms, we minimized the effort necessary to implement an energy-aware policy for a complex underlying hardware architecture. We moved considerable computational overhead from runtime to the design phase, rendering the development of a sophisticated energy model, featuring dozens of energy weights per core, to solve the problem unnecessary.

6.3 Future Directions

The power and performance-related analysis of chip multi-processors is a difficult task, given the complexity of the underlying hardware and the number of possible configurations and dependencies between software execution, hardware constraints and the mutual influence in each case. In order to focus the scope of this thesis on methods immediately leading to a gain in energy efficiency, we excluded an entire class of scenarios from our analysis, namely the possibility of task co-scheduling once more active tasks exist in the system than processor cores.

In server and high-performance computing environments, systems have to run at peak efficiency while under full load or even when oversubscribed with processing jobs. In these situations, task-specific knowledge can be instrumental in order to facilitate optimized scheduling decisions compared to a scheduler that is oblivious to the energy-related characteristics of a task. These scheduling strategies especially have to take shared resources such as limited memory bandwidth into account. Although this thesis only considered configurations with a limited number of concurrently active tasks, our results can also be applied to the overutilized case. The presented bandwidth model for the given quad-core processor architecture can derive the actual memory-intensiveness of a task depending on the overall bandwidth situation. This type of edited data promotes more meaningful energy-related task profiles and may improve the accuracy of advanced co-scheduling policies – such as *memory-aware scheduling* [19] and *task activity vectors* [20] – striving to avoid resource contention due to the accompanied negative impact on energy efficiency by influencing the order runqueues are processed. In conjunction with energy-efficient frequency scaling, as presented in this thesis, it is possible to increasingly harness the full efficiency of the processor's hardware.

As the underlying processor architectures evolve, also these software-based techniques will. For the future, it is probable that some of these energy-aware mechanisms will eventually find their way into commodity operating systems or even the hardware itself, as the need for efficient computer systems is becoming inevitable.

List of Algorithms

Index

Bibliography

[1] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *In Proceedings of the 9th ACM SIGOPS European Workshop*, 2000.

[2] Frank Bellosa. *Three Dimensions of Scheduling.* PhD thesis, University of Erlangen-Nürnberg, Germany, November 27 1998.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-232-8.

[4] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED*, pages 174–179, 2004.

[5] R. Choi and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to onchip computation times. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, January 2005.*, 2005.

[6] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies, and Toshiba Corporation. *ACPI specification 3.0a*, December 30 2005.

[7] Intel Corporation. *Mobile Intel Atom Processor N270 Single Core*, 2008.

[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*, 2008.

[9] Intel Corporation. Introducing the 45nm next-generation Intel Core microarchitecture, 2007.

[10] Alexandra Fedorova. *Operating System Scheduling for Chip Multithreaded Processors.* PhD thesis, Harvard University, September 2006.

[11] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for Linux. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 105–116, New York, NY, USA, 2002. ACM.

[12] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, vol. 30, no. 9:79–85, September 1997. Stanford University.

[13] John L. Hennessy and David A. Patterson. *Computer architecture*. Morgan Kaufmann, 3. ed. edition, 2003. ISBN 1-55860-724-2, 1-55860-596-7.

[14] Simon Kellner. Event-driven temperatur control in operating systems. Study thesis, Operating System Group, University of Erlangen, Germany, April 30 2003.

[15] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson. Characterizing the impact of different memory-intensity levels, 2004.

[16] Evangelos Koukis and Nectarios Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of SMPs. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 345–354, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2612-8.

[17] Min Yeol Lim and Vincent W. Freeh. Determining the minimum energy consumption using dynamic voltage and frequency scaling. *Parallel and Distributed Processing Symposium, International*, 0:348, 2007.

[18] Andreas Merkel. Balancing power consumption in multiprocessor systems. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, September 30 2005.

[19] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *HotPower Workshop '08*, 2008.

[20] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 1–12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5.

[21] Trevor Mudge. Power: A first-class architectural design constraint. 2001. University of Michigan.

[22] Yiannakis Sazeides, Rakesh Kumar, Dean M. Tullsen, and Theofanis Constantinou. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Computer Architecture Letters*, 4(1), 2005. ISSN 1556-6056.

[23] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal*, 11(4):361–370, 2007. ISSN 1535-864X.

[24] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 84–93, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-825-1.

[25] Andreas Weissel and Frank Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246, New York, NY, USA, 2002. ACM. ISBN 1-58113-575-0.

[26] Dominik Winkelmeyer. CPU power management for SMT and CMP processors. Study thesis, System Architecture Group, University of Karlsruhe, Germany, March 21 2007.